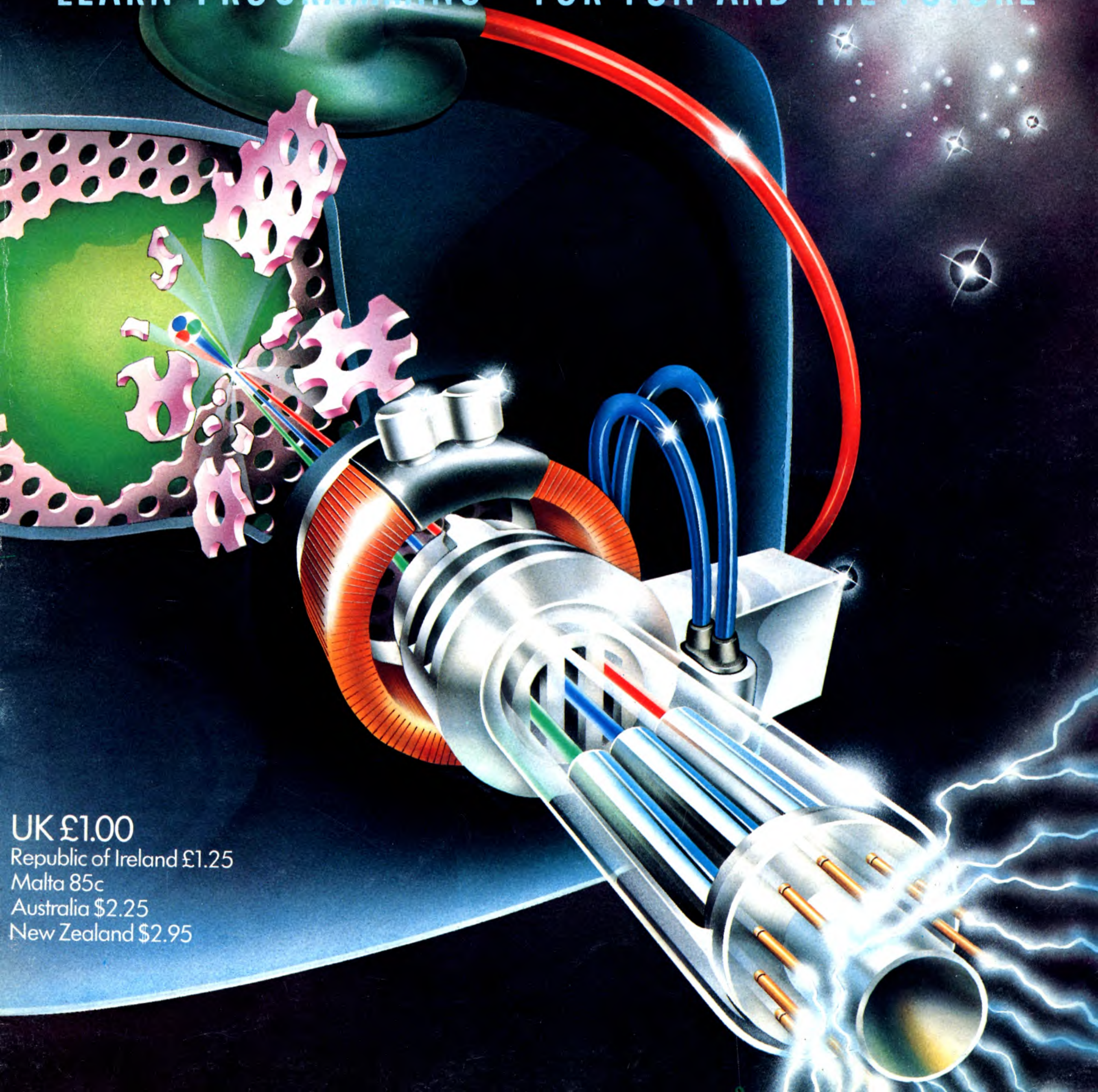
A MARSHALL CAVENDISH **15** COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 2 **No 15**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS

Front cover, Kuo Kang Chen. Page 445, Malcom Harrison. Page 446, Kuo Kang Chen. Page 448, Stan North. Pages 450, 452, 454, Artist Partners/Gary Keane. Page 458, 461, Graeme Harris. Pages 464, 466, 469, Chris Lyon. Page 470, 472, 475, Digital Arts.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

There are four binders each holding 13 issues.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to: *Marshall Cavendish Partworks Ltd.*

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

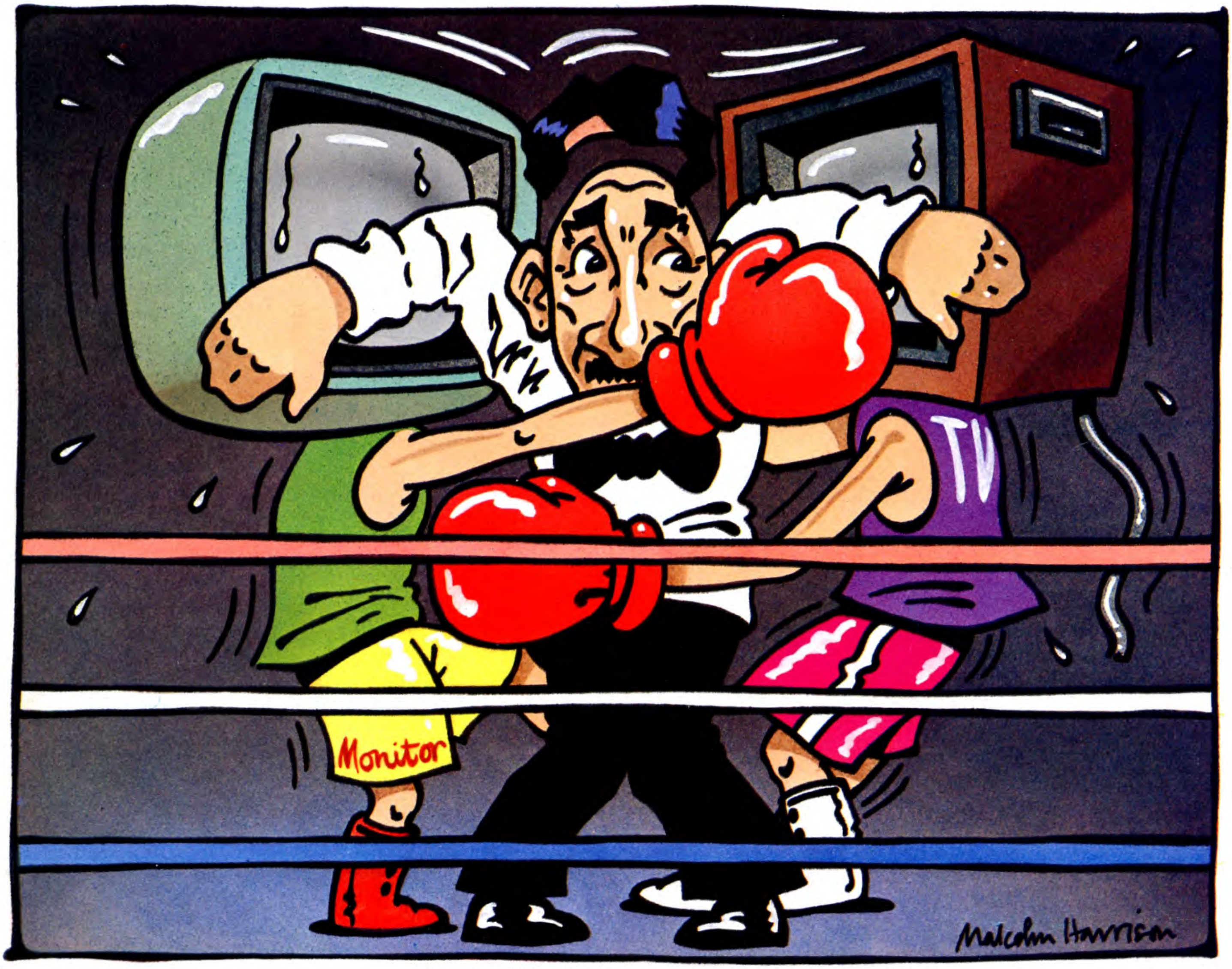
The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64. .

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

TV VERSUS MONITOR

- HOW TVS AND MONITORS WORK
- TYPES OF SIGNAL INPUT
- COLOUR VERSUS MONO
- PICTURE SHARPNESS
- MAKING A CHOICE



Making a choice between a quality TV and monitor can be difficult, but for anything but casual use there can be only one option. Here we look at some of the general differences between the two types

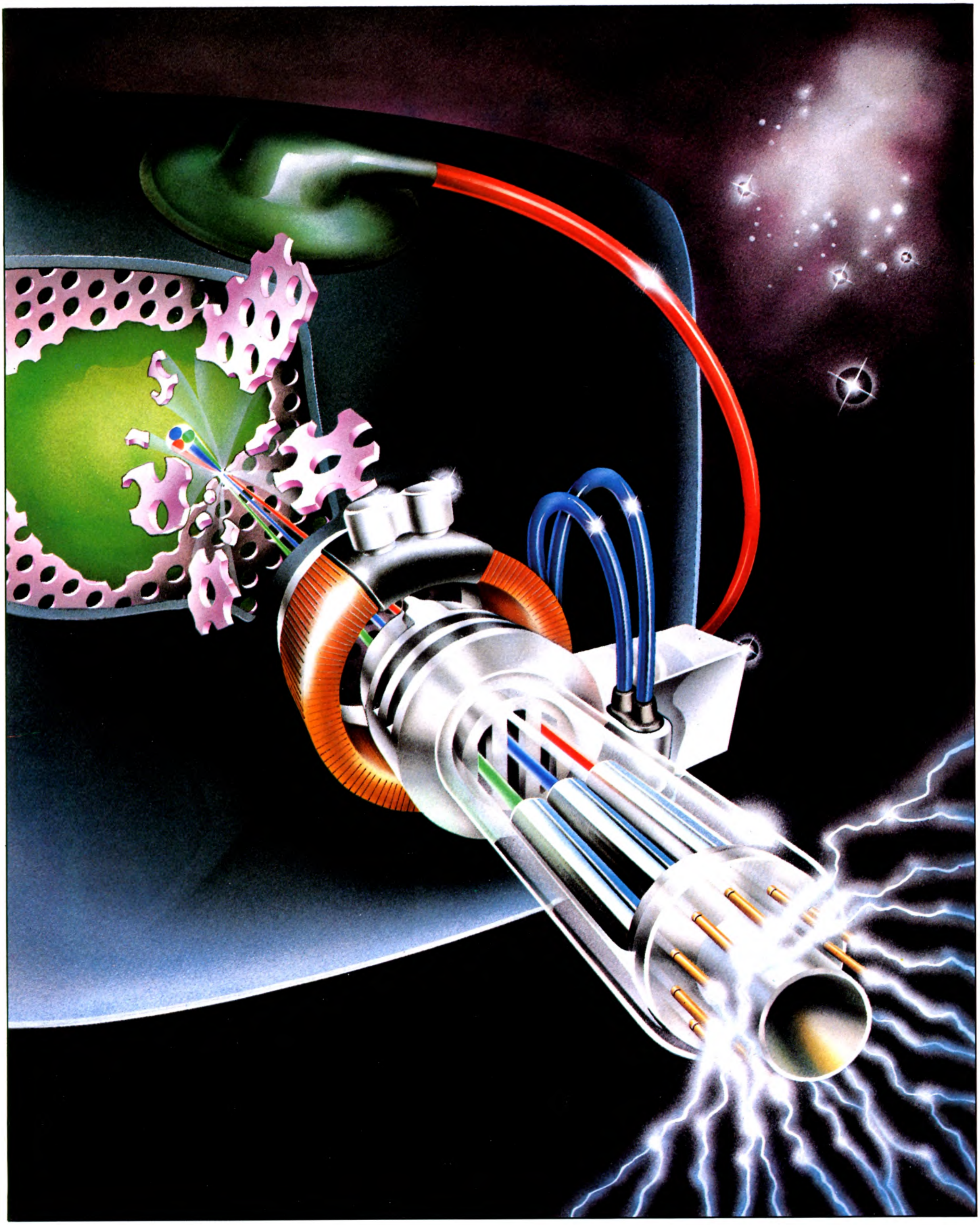
The screen is a vital part of any computer system, even if in many respects it is seen as more of a workhorse than a sophisticated piece of electronics. The screen is the main channel through which the computer communicates with us. Information flows the other way, from user to computer, usually through the keyboard and the screen also displays a record of the information that's been typed in.

Whether you are in the fortunate position to be able to choose a new screen, or just want to get the best out of what you have, it is worthwhile—if not essential—to know something about how a screen works and how a computer controls video output. This in particular will affect your choice.

FIRST PRINCIPLES

Although there are several different ways in which information is transferred from the computer to the screen, there is one feature common to all types of screening used in business, industry and home—the cathode ray tube, otherwise known as a CRT. Cathode ray tubes work because certain phosphor compounds glow when hit by a stream of electrons.

Fire a stream of electrons from a gun at a sheet of glass coated with phosphor and



wherever the gun is aimed the phosphor will glow. Sweep the gun back and forth across the sheet of glass so that the whole screen is bombarded by electrons and the whole screen will glow. Wave the gun in a figure of eight and a figure of eight will appear on the screen.

A cathode ray tube is a vacuum-filled glass cone with an electron gun at one end and a base coated with phosphor.

Imagine a piece of card with the shape of a man cut out of it, held between the gun and the screen. The screen would only glow where the stream of electrons was not blocked by the card. Consequently the area of phosphor glowing on the screen would resemble the shape of a man. The same effect could be achieved by switching the electron gun on and off at the right times—when it is pointing to the desired spots on the screen.

This is exactly how a TV works and how all but a few specially-built monitors work. An electron gun sweeps back and forth across the screen 625 times to create 625 lines. (In fact it sweeps across 313 times first and then fills in the 312 gaps between those lines.)

There are no mechanical parts, the aiming of the stream of electrons is done by using magnetic fields. That's why it's very unwise to let any magnetic media—tapes or disks—get anywhere near a TV set. An image is formed by switching the electron gun on and off. All that the signal received by the TV or monitor does is tell the electron gun when to switch on and off.

Rather than sweeping across the screen in what is called a *raster scan*, a few monitors produce a picture by guiding the electron beam around the screen a little like a pencil, following the lines of the image. Although this produces excellent quality results it is much too slow for general use. Even with the electron beam moving across the screen at the normal 25,000 or so miles per hour, the human eye would find a picture of any complexity produced by this method highly unsatisfactory.

PERSISTENCE OF VISION

It is one of the characteristics of the human eye which might, in other circumstances, be considered a failing that enables us to use the cathode ray tube to communicate visually.

The human eye retains any image that hits the retina for about one twenty-fifth of a second. The electron gun at the back of the cathode ray tube inside the vast majority of TV sets and monitors sweeps back and forth across the screen 625 times every fiftieth of a second. This means that a new picture or 'frame' appears on the screen fifty times a second. A new image appears long before the old one has faded from our eye. Therefore we see an image that moves smoothly.

If our eyes retained an image for only one hundredth of a second we would see an image that flickered very badly.

COLOUR

The principle is the same for both monochrome and colour sets. The only difference between them is that only one electron gun and one kind of phosphor coating is required for mono sets while colour sets require three different sorts of phosphor coating and usually three different electron guns (there are sets with just one gun, but firing three streams of electrons).

Colour is produced because one coating of phosphor glows red, another green and the other blue. One stream of electrons activates the red phosphor, another the green phosphor and the third the blue phosphor. By combining red, green and blue in different strengths and combinations, all other colours and intensities can be built up.

SIGNAL INPUT

Screens, whether TVs or monitors, colour or black and white, can accept three kinds of input. First of all, there is the standard broadcast signal, the sort of signal accepted by TV sets and which produces standard TV pictures on the screen. Then there is the input which is usually called *composite video*. This signal is the one most commonly used for monitors and there are now some TV sets which have a facility for accepting these signals. The signals contain picture information plus synchronization pulses.

Finally there is *RGB input*. The letters stand for Red, Green and Blue and it is the most direct and, therefore, accurate, form of colour input. Information about each colour is fed from the computer to the monitor separately.

The composite video input is a halfway house between broadcast and RGB input. It is, however, much better than the broadcast TV signal. This signal must first of all be translated inside the TV from an electrical signal. Inside the TV the signal needs to go through more processes involving modulation and amplification before it becomes a picture. Obviously the more stages a signal needs to go through, the greater the chances of noise and distortion.

> **Microtip**
>
> **Viewing conditions**
>
> With both a monitor and TV you can do much to optimize viewing conditions:
>
> You can reduce eyestrain by using light coloured characters against a black background. To improve sharpness, play about with the contrast and brightness controls. You can usually eliminate colour fringeing simply by turning the colour signal down or off—after all, you don't require colour for word processing.
>
> Try to position your work desk—or TV/monitor itself—so that the screen does not reflect lamplight or window light. Sometimes you may find wearing dark clothing a help. You can further minimize the effects of reflections by looking slightly down at the screen rather than keep it square and at eye level or higher.

BANDWIDTH

There are some interesting but complicated technical reasons why a TV set cannot give such good resolution as a monitor. A TV set, for instance, cannot handle information which requires a *bandwidth* more than 5.5 MHz. This is fine for a 40 column display but for satisfactory definition in 80 column displays, a bandwidth in excess of 10 MHz is required.

Monitors are specifically designed to accept such a bandwidth. The input for most monitors available for the home computer market is a composite video signal and a monitor does not require the tuner and modulator necessary in a TV set. Because it needs much less processing, the composite video signal from a computer is usually much 'cleaner'.

When people talk about a 'good' picture on a screen they are usually referring to resolution and colour. Unfortunately the two aren't always compatible. A monochrome screen will always give higher picture definition than a colour screen. Only one dot of phosphor is required to define a point on a monochrome screen, while three similarly sized dots would be required to define the same point on a colour screen.

On a normal screen of any kind there are something like 360,000 dots of phosphor. In the case of a colour screen 120,000 of these dots will need to be red, 120,000 will need to be green and 120,000 will need to be blue. This means that there are only 120,000 units consisting of three dots available to make a picture on a colour screen. But a mono screen has 360,000 dots available to create an image.

Anyone who has worked both with a monochrome monitor and a colour TV will know the difference. The monochrome monitor is much

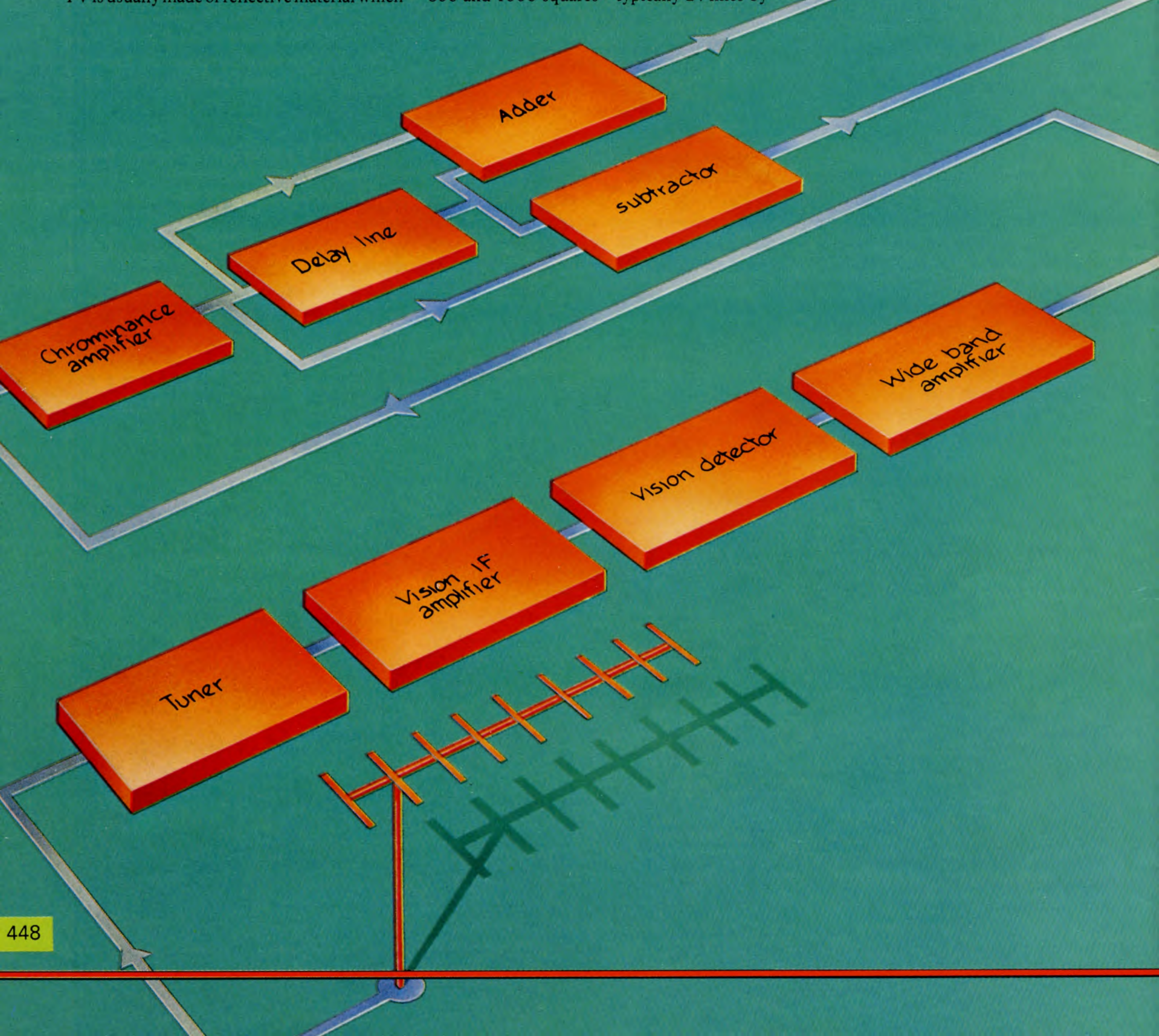
sharper and clearer while the colour TV gives what is a comparatively blurred picture.

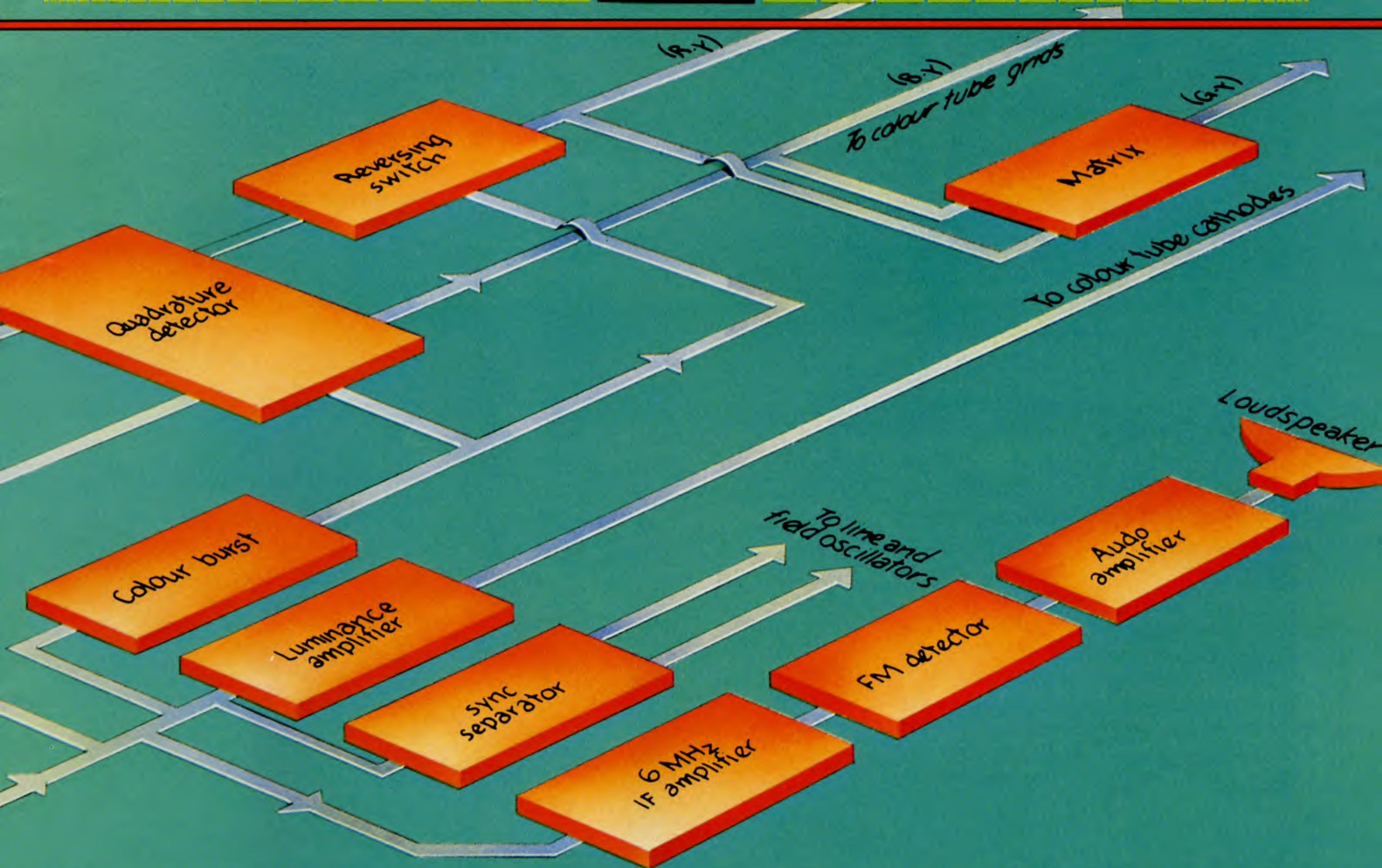
For many specialist applications a monochrome monitor will be the natural choice. The usual colour chosen for such applications—computer aided design or wordprocessing, for instance—is green. This is reckoned to be the colour that's easiest on the eyes.

On a health note, the vast majority of reports that have every been published pass the cathode ray tube as completely safe for the eyes. It's usually found that complaints arise from other working conditions or are caused by unsuitable screen colours, glare from the screen, distraction from other light sources, and sometimes psychological factors. But a TV screen does flicker slightly and the screen of a TV is usually made of reflective material which often shows unwanted and distracting images. This is one of the reasons why using a TV for serious work requiring concentration over long periods may cause headaches.

In order to produce a picture with the resolution of ordinary TV or video pictures, our computers would need to be able to control every single dot of phosphor on the screen. And if each pixel requires a byte of memory, half a megabyte of memory would be necessary to control the screen. This is obviously out of the question as far as home computers are concerned and would be a tremendous waste in most applications of bigger, business and research computers.

Most home micros allocate about 1K of memory by dividing the screen into between 800 and 1000 squares—typically 24 lines by





40 columns. Others give you a choice between low resolution and high resolution depending on the chosen display modes. Note, however, that this has nothing to do with the screen you are using. Limitations in picture definition and resolution are imposed by the computer and not by the CRT.

Each one of these 900-odd squares is an individual character cell. The computer starts off with a character set in ROM which uses an ASCII number to store each character. When they are printed on the screen each character takes up one square. In this way each character can be stored in a single byte, representing a huge saving in memory.

MAKING A CHOICE

The two most important factors to consider when buying a screen are the type of computer, and future applications. Although the screen is an important element in the system it always remains dependent on the computer. If the graphics capabilities of the computer are poor, then they will remain poor no matter how good the screen. The computer is always in control of the screen and it's important that the screen is bought to suit the computer.

It's also important to consider what the computer will be used for. For the best results in applications where colour is never necessary, a monochrome monitor is a must. If high quality colour is essential then a monitor with RGB input is ideal although even most professional users of micros would usually only require composite video input. For owners of all but the most expensive micros the choice between RGB and composite video monitors will remain academic for some time to come since few micros are built with RGB output, the Acorns being one exception.

Games, general 'fun' programming at home and other less serious uses of computers in the home only demand an ordinary TV set. Though many games would look much better on higher definition screens others are written in the full knowledge that their graphics will never be challenged by anything other than an ordinary TV set. Remember, too, that because monitors are manufactured mainly for the business market they often don't include such facilities as sound!

Even though they are often electronically simpler, monitors aren't necessarily cheaper than television sets of similar size. As far as screens are concerned, bigger is not necessarily better. In many cases it might be a lot worse. All that happens as the size of the screen increases and decreases is that the size of the individual pixels increases and decreases. If the computer produces 960 pixels then that's how many pixels there will be on the screen no matter how big or small. There isn't usually any increase in the amount of information available on the screen and all that happens in some cases is that a bigger screen makes the rough edges even more noticeable. Incidentally, remember that screen sizes are taken across the diagonal and do not represent the width of the screen.

Old or very cheap TV sets are not recommended. Often the problems which don't seem at all serious when a picture produced from an ordinary broadcast TV signal is displayed on the screen can be disastrous when the set is used in conjunction with a computer.

Overscanning, for instance, which cuts off the sides or the top and bottom of the picture, may not seem too serious when pictures are displayed, but could mean that valuable information from the computer does not appear on the screen. Another fault of old TVs is 'blooming', when one colour becomes much stronger than the others producing a blurred effect which can make type on the screen very difficult to read. Usually repairs to these faults prove expensive—it's often cheaper to buy a better TV . . . or a monitor!

MAKE MORE OF YOUR UDGS

Versatile, varied—UDGs are what you make them and are limited only by your imagination. But to get the most out of them, you may need to use special techniques

UDGs have other uses far more varied than creating just another alien. For instance, you might want to use them to create a set of special symbols or letters, with accents for such as è, (page 43 shows you how to do this for to do this for the Commodore 64). Later on in this article you'll find how to create a variety of new UDGs. But first you may need to get around the limitations of your computer.

LIMITS ON UDGS

Creating UDGs is all quite simple and easy to do, but unfortunately, some computers limit the number of user-defined graphics characters. The Acorn machines limit you to 32 characters, the Spectrum to 21, while the Commodore machines are less likely to pose problems, with 256 available. The Dragon and Tandy have no actual UDGs, but let you use variable arrays to store details of your characters—and so the only limit is the number of variables, which is as good as limitless. The ZX81 has no UDG facility at all.

If you have a Spectrum or Acorn computer, there are lots of occasions when you might want to use more than just 21 or 32 UDGs, and so these limits seem rather inhibiting.

For example, you might want to draw a screen display using large numbers of UDGs—maybe a Frogger-type game, with several different cars and lorries, motorbikes, logs, alligators, and, of course, a frog. By the time that you have designed all of these, you will almost certainly have used more than 32 UDGs; and you still have to produce some background such as river banks, or a road side.

Or suppose that you wanted to draw a picture of a city street. You would need UDGs for the buildings, for the people, for the cars and other vehicles, and for any other details that you might want to include. The next article will show you how to create a complete picture.

Finally, you might well want to create an extended range of characters,—the Greek alphabet, for example—which would exceed the standard number.

In other words, there are numerous occasions when you might like to have more than



- HOW MANY UDGS CAN YOU DEFINE?
- CREATING SPACE FOR CHARACTERS
- FINDING A SAFE PLACE IN MEMORY FOR THE DATA
- RESETTING CHARACTER SET ON THE COMMODORE AND SPECTRUM
- CREATING NEW CHARACTERS
- REDEFINING THE NUMBER SET



your computer's limit of UDGs. Luckily, there are ways to increase the number possible. Each computer allows this differently.

The Spectrum has space for 21 UDGs already reserved when you turn it on, but you can have more by clearing an extra area in RAM specially for them. What you do once the extra UDGs are present in memory is to have several 'banks' of 21 UDGs, which you use as and when you want.

HOLDING THE DATA

The first step in the process is to find a secure place in memory for the DATA, which can't be corrupted by a BASIC program. You must then decide how many UDGs you want to store in RAM so that you know how much memory to reserve.

Suppose that you wanted 21 extra UDGs. You can find out the length of the block of memory you need by multiplying 21 (the number of extra user defined graphics) by 8. The 8 is used since 8 bytes of memory define every character. The result of this sum gives you the number of bytes you need to reserve.

The next thing to do is to work out the best place to store the DATA. The higher up in memory you store it, the more space you will have left over for your BASIC programs. It's best to put it as high as possible, immediately beneath the UDG area already there, as this is as high as a BASIC program can go.

This address is known as RAMTOP. RAMTOP is 1 less than the first byte of the UDG area when you turn your Spectrum on. You will see later that this may not always stay the same, so you can find out where RAMTOP is by entering the following command:

```
PRINT USR "A"—1
```

As you can see, this PRINTs the address of the first byte in the user defined graphics character "A" less 1. If you haven't altered RAMTOP this address should be 65367 for 48K Spectrum and 32599 for 16K machines.

Now that you know where RAMTOP is, you can work out the start address for your UDG banks. RAMTOP is the last address that you can use for a BASIC program: all the memory above it is out of bounds to BASIC programs.

Next you must work backwards. Suppose that you want to have one extra bank of UDGs. As a bank of UDG DATA is 21*8 or 168 bytes long you must start your DATA 168 bytes before RAMTOP. This is at location:

```
USR "A"—169
```

which is at location 65199, or 32431 on 16K Spectrums.

PROTECTING THE DATA

Putting your DATA here is not much help if a BASIC program strays into the same area to change it, which could happen if you wrote a very long program. But if you type in (or have a line in your program which says):

```
CLEAR USR "A"—169
```

then RAMTOP is lowered by 168 bytes and the area from 65199 upwards (or from 32431 on a 16K Spectrum) will be reserved, and protected from being overwritten by BASIC.

Now you have to tell the Spectrum where to find the DATA, and then actually store the bytes of your characters in memory.

USING THE POINTERS

To tell the computer where the DATA for the extra UDGs is you need to change a *pointer*. There are a number of pointers in your Spectrum's memory, and one is used to point to the address of the first byte of your UDGs. When the computer wants to use the DATA stored away, it PEEKs the pointer, which points it to the correct address. This is normally set at 65368, but you'll need to alter this to point to your new bank of UDGs.

One feature of pointers is that you can POKE new values into them, so that the computer will look at a different address, and it is this that makes them so useful for extra UDGs. In fact, you need two POKEs to change the address of a pointer, because one byte on its own can only hold a number as large as 255, while most address numbers are larger than this. So, the Spectrum splits the number into two parts, the equivalent of the tens part and the units part of a normal number.

To find the different parts of a number, you divide by 10 if it is a two digit (and decimal) number. For instance, take the number 56. To find out how many tens there are, you divide 56 by 10, to get the answer 5. The remainder, in this case 6, becomes the second part of the number.

When you POKE an address into the pointer, you can split the address into two parts in a very similar way. The difference is that instead of dividing by 10, you divide by 256. For the 21 extra UDGs this means starting your extra DATA at address 652ØØ. So you need to POKE 652ØØ into the pointer in two separate parts. Using the method described above, you divide by 256. This gives 254 with a remainder of 176.

The Spectrum's system takes the first number of any two-part numbers as being the low (rather than the high, which the human brain is used to) part. So the first number that you work out actually goes into the second part of the pointer, and the remainder goes into the first part.

The two parts of the pointer have two addresses. For the UDG pointer, these addresses are 23675, and 23676. So the POKEs you need are:

```
POKE 23675, 176
POKE 23676, 254
```

All that's left now is to define and input the DATA for your extra characters.

Once you've changed the UDG pointer, you do exactly the same as you would for normal UDGs: you POKE USR "A" with the first byte of your first character, USR "A" + 1 with the second byte, and so on.

If you like, you can POKE the DATA in before you change the pointers. In this case you would POKE each byte directly into the memory, so the first byte goes into location 652ØØ and so on. So as not to get confused with addresses just add 1 to the address for every extra POKE.

However, you will need to change the UDG pointer to be able to use the DATA for UDGs, anyway, so it's usually easier to change the pointer at the start.

Using the UDG facility of your Spectrum like this is all very well if you can make do with printing the characters from within a program. If you need to have the characters accessible from the keyboard this method is less suitable. You may not want to keep changing the pointer, or using graphics mode—and it is certainly very fiddly to use CHR$ all the time.

A NEW CHARACTER SET

You can get around this by redefining the Spectrum's character set. This means that pressing a key will print your UDG rather than the normal letters or numbers.

You might want to redefine the character set for a number of reasons. For example, you might want to print messages on the screen in a foreign language: Russian or Greek, perhaps. Or you might just want to 'personalize' your programs with your own specially designed typeface.

Redefining your Spectrum's character set is actually very similar to getting extra banks of UDGs: you reserve an area of memory in the same way, and POKE in the DATA in a similar way as well.



There is one problem with changing the character set: if you just want to change some of the characters, you still have to transfer the DATA for the rest of the set into RAM for the Spectrum to use.

The reason for this is that you need to change the character set's pointer, in a similar way to the pointer for the UDGs. Once the pointer has been altered the computer looks for the character DATA at the address you have given it, it cannot return to ROM for the characters you want to leave unchanged.

You can overcome this by placing the whole of the DATA for the ROM character set in RAM before you start putting in your own DATA for the characters you want to change.

First type RANDOMIZE USR 0 to reset the memory then enter the next program:



```
10 CLEAR USR "A"-769
20 LET d=PEEK 23730+256*PEEK 23731+1
30 FOR n=15616 TO 15616+767
40 POKE d,PEEK n
50 LET d=d+1: NEXT n
60 POKE 23606, PEEK 23675
70 POKE 23607,PEEK 23676-4
80 LET p=PEEK 23606+256*PEEK
   23607+48*8
90 FOR n=p TO p+79: READ a: POKE n,a:
   NEXT n
110 DATA 0,124,76,84,86,102,126,0
130 DATA 0,8,8,8,24,24,24,0
150 DATA 0,126,2,126,96,96,126,0
170 DATA 0,124,4,126,6,6,126,0
190 DATA 0,96,98,98,126,2,2,0
210 DATA 0,124,64,126,6,6,126,0
230 DATA 0,62,32,126,98,98,126,0
250 DATA 0,124,4,4,6,6,6,0
270 DATA 0,60,36,60,102,102,126,0
290 DATA 0,124,68,126,6,6,126,0
```

The first eight lines protect an area of RAM from being overwritten by BASIC, and place the DATA from ROM into the reserved area of RAM. The remaining lines redefine the numbers. RUN the program, wait until the message 'OK' appears on the screen then press any of the number keys.

You can NEW the program at any time, and the computer will revert to the ROM character set. The defined set is still in RAM, however, and can be 'switched in' by entering:

```
POKE 23606, PEEK 23675:
POKE 23607, PEEK 23676-4
```

It does not matter whether you have a 16K or a 48K Spectrum—the program checks to see, and POKEs the relevant addresses. It does this by PEEKing the RAMTOP pointer (this is at addresses 23730 and 23731), and combining the two PEEKs together in such a way as to produce the address of RAMTOP.

The ROM character set is stored at addresses 15616 to 15616+767 (at least, these addresses hold the DATA for that part of the character set that you can redefine). So using a FOR ... NEXT loop the program POKEs in the contents of each byte in this area of ROM to the corresponding byte in your newly reserved area of RAM.

The loop automatically updates the address in ROM that it PEEKs (since the control variable of the loop is also the address that is being PEEKed) and the address that is being POKEd (variable d) is updated in Line 50. As you can see from Line 30, there is no need to add up all the addresses and byte numbers yourself, since the Spectrum can do it for you. Lines 60 and 70 update the pointer for the character set so that it now points to the start address of the DATA in RAM. The address is found by PEEKing the address of the start of the UDG area. (This works even if you reserve extra banks of UDGs.) The character set in RAM is stored immediately beneath the UDGs, and so the calculation in Line 70 simply deducts the length of the character set from the address of the UDG area. The length is 4, but this is the high byte of the address, so you have to multiply by 256, and 4*256=1024 bytes.

Now that the character set has been copied into RAM, you are ready to replace some of the old characters with your new ones.

Unless you are careful, you might replace the wrong characters, which could produce some unwelcome results. You can prevent this by working out which of the characters already there you want to change, and by POKEing your new DATA into the right places.

If you look at Appendix 1 in the Spectrum manual you will find a listing of the character set. You can only redefine the ASCII characters with codes from 32 to 127 (the left hand column in the appendix gives the code for each character).

To find out which bytes you need to change multiply the code of the character by 8. This gives you the number of the first byte of the first character that you want to redefine. For the space character (ASCII code 32), this is $32 \times 8 = 256$.

Now that you know how far into the character set the byte is, you can simply add this number to the address held by the character set pointer. This works out as PEEK 23606+256*PEEK 23607+32*8. And if you've just turned on your 48K Spectrum, the address is 64600.

Giving the Spectrum the DATA for the character is straightforward after this. You just POKE eight bytes, starting with the one whose address you have just found, with the numbers that represent your character. These few lines redefine the space so it looks like one of *INPUT*'s space boxes. It POKEs in new DATA using a FOR ... NEXT loop:

```
10 FOR X=64600 TO 64600+7
20 READ A
30 POKE X,A
40 NEXT X
50 DATA 0, 126, 66, 66, 66, 66, 126, 0
```

If you own a 16K Spectrum, change the 64600 to 31832. You can see that it is a good idea to use a FOR ... NEXT loop to POKE in the different bytes, as the loop changes the address for you, and saves you having to do eight separate POKEs.

Try to work out what address the number characters begin at: you can see whether or not

you are correct by looking back at the program that redefined them. The program works even if you have moved the UDG area around, and the address is given by P in Line 8Ø.

The Commodore lets you define your own characters using several POKE commands, as the article on pages 38 to 45 showed.

The program on page 43 and the one below disengage the Commodore's own character set and tell the computer to look for the character set in RAM, at a place where the program has stored some replacement DATA. This example redefines the numbers from Ø to 9 but note that it takes two minutes before you can see any result:

```
10 POKE 52,48:POKE 56,48:CLR
20 POKE 56334,0:POKE 1,35
30 FOR Z=0 TO 4095:POKE
   12288+Z,PEEK(53248+Z): NEXT Z
40 POKE 1,39:POKE 56334,1
50 FOR Z=0 TO 79:READ X:
   POKE 12672+Z,X:NEXT Z:
   POKE 53272,28
100 DATA 0,124,76,84,86,102,126,0
110 DATA 0,8,8,8,24,24,24,0
120 DATA 0,126,2,126,96,96,126,0
130 DATA 0,124,4,126,6,6,126,0
140 DATA 0,96,98,98,126,2,2,0
150 DATA 0,124,64,126,6,6,126,0
160 DATA 0,62,32,126,98,98,126,0
170 DATA 0,124,4,4,6,6,6,0
180 DATA 0,60,36,60,102,102,126,0
190 DATA 0,124,68,126,6,6,126,0
```

The snag with this method is that you can only define your own characters at the expense of characters already in the computer. So if you are replacing them completely, the trick is to change some of the less useful characters.

This is quite straightforward. If you look at Appendix E of your Commodore manual, you can see a list of all the characters that are stored in ROM. There are two sets of characters, upper and lower case. Each of the sets is actually stored twice in ROM: once for normal characters, and once for reverse characters.

If you look at set two, you will see that there are a number of gaps. These indicate that the character is the same for both upper and lower case; but the DATA is stored twice. So, by choosing characters which are the same in both sets, you can define some characters of your own without losing any of the standard characters.

WHERE TO PUT THE DATA

Once you have chosen the characters you must calculate the addresses of its bytes in RAM. Suppose that you want to redefine the lower case @ character (this is the first



character that is the same in both upper and lower case). Take the POKE code of the character (in this case Ø) and multiply it by 8. The result, here Ø, is the position of the first byte in the character set.

Since this character is a lower case one, (that is, it is in set 2) you add the length (in bytes) of the first character set (256*8, or 2Ø48) to the number you already have. This gives you the number of the first byte that you want in the whole character DATA. In this case 2Ø48 + Ø = 2Ø48.

The start address of the character set in RAM can be one of 6 positions when used with BASIC program, but it is usually 12288.

Now add these two numbers to give the address of the byte you want to change.

All you do now is POKE the address with the DATA for the new character—you can see how to work out the DATA values by looking at the article on pages 38 to 45.

After you have POKEd the first byte of the character, you find out the address of the next by adding one to the first address; you do this for each of the other seven bytes (there are eight bytes in every characters).

If you want to redefine more than one character choose one which is the first of several that are the same in upper and lower case, you can save yourself having to calculate all the different addresses.

You should always check your calculations very carefully: if you get even one wrong you will corrupt other characters and you might be faced with a strange assortment of characters when something is PRINTed on the screen.

If this does happen, you needn't turn the computer off and start again. All you need to do is POKE the character set pointer back to its usual address:

```
POKE 53272, 21
```

Alternatively, you can press RUN/STOP and RESET, which resets the pointer to its usual value. (If you press RUN/STOP and RESET at any time, whether you want to return to the original character set or not, the Commodore automatically re-engages its own set of characters.)

If you use both upper and lower case character sets you can have up to 512 of your own characters available at any one time. Should you want more than this for any reason, you can store several sets of characters at different places in memory, and then all you have to do is change the character pointer to the relevant address when you want to use a different set.

The 7 possible places you can store the UDG DATA are:

2Ø48 4Ø96 6144 8192
1Ø24Ø 12288 and 14336





The Spectrum's numbers are on the top line, the newly designed computer-style numbers underneath. You could redesign the whole alphabet in a similar way.

If you really want to use a lot of your own graphics characters, you can use up to four of the addresses, and have four separate banks of UDGs. The main problem with this is that you then have very little space left for a BASIC program.

As each bank of character DATA takes up 4K of memory, you cannot use every address (if you intend to fill up each area), as the addresses are only 2K apart, and you can only have the maximum of 4 if you start with first address, 2Ø48.

When you use the first area to store the bytes which make up your characters, you must move the start of BASIC up in RAM. BASIC normally starts at 2Ø48, and so it would be in the way of your characters.

You quite easily can work out how far you need to move the BASIC area. Each character requires eight bytes. So if you have a hundred characters to define you will need to clear 1ØØ*8=8ØØ bytes of memory. Add this to the start address (2Ø48) to get the address of the end of the UDGs—2848 in this case.

Moving BASIC around in RAM is an easy matter on the Commodore 64. All you need to do is to POKE three locations, then type NEW and hit RETURN. Locations 43 and 44 hold the start address of BASIC and these have to be POKEd with the new address. Each location can only hold a single byte, that is any number up to 255 and since most addresses are larger than this, the address has to be divided into two parts. Say you want BASIC to start at 19ØØØ. All you do is divide by 256 to give 74 with a remainder of 56. Then POKE the high byte (74) into location 44 and the low byte plus 1 (57) into location 43.

You also POKE the start address of BASIC, here 19ØØØ, with Ø:

```
POKE 44, 74
POKE 43, 57
POKE 19ØØØ, Ø
```

Now type NEW, and a series of other pointers will be changed to match the new start address of BASIC.

You are now able to type in a BASIC program again, and have up to four sets of characters in memory.

To bring one of the various banks of UDGs into use you just change the character pointer to point to the address of the character set that you want to use. The character pointer is at address 53272.

Unfortunately, you cannot just POKE in the actual address of your character set: you must change it into a form that your Commodore can understand.

You can work out the number you have to POKE into the pointer by entering a direct command like this:

```
PRINT (PEEK (53272) AND 24Ø)+14
```

This is for the last bank of characters at address 14336. Don't worry about the first part of the Line, it is the last number, 14, which is the important part, and changes according to what address you want to convert. Its value can be worked out easily—you just divide the address by 1Ø24 and use the result. For example, suppose you want to engage the set of characters at address 6144. You replace the number 14 above with 6. Similarly, if you want to 'switch in' the characters stored from address 1Ø24Ø, you replace the 14 with 1Ø.

When you have entered the command it will print out a number—in this case it gives 3Ø. Once you have found the answer, you simply POKE 53272 with 3Ø. Once the computer has executed this command, it uses your new characters, and even characters already on the screen are changed into the new ones.

This updating of the screen is a mixed blessing. It can be used to produce interesting effects, but it also means that you cannot increase the number of UDGs available on the screen by PRINTing something in one set, changing sets, and then PRINTing again.

You can define your own characters on the Vic by changing the standard ROM character set, as on the Commodore 64.

Type and RUN this short program which shows just one way that you can use the UDGs:

```
1Ø POKE 52,2Ø:POKE 56,2Ø:CLR
3Ø FOR Z=Ø TO 2Ø47:POKE
   512Ø+Z,PEEK(32768+Z): NEXT Z
4Ø FOR Z=Ø TO 79:READ X:
   POKE 55Ø4+Z,X:NEXT Z
5Ø POKE 36869,253
1ØØ DATA Ø,124,76,84,86,1Ø2,126,Ø
11Ø DATA Ø,8,8,8,24,24,24,Ø
12Ø DATA Ø,126,2,126,96,96,126,Ø
13Ø DATA Ø,124,4,126,6,6,126,Ø
14Ø DATA Ø,96,98,98,126,2,2,Ø
15Ø DATA Ø,124,64,126,6,6,126,Ø
16Ø DATA Ø,62,32,126,98,98,126,Ø
17Ø DATA Ø,124,4,4,6,6,6,Ø
18Ø DATA Ø,6Ø,36,6Ø,1Ø2,1Ø2,126,Ø
19Ø DATA Ø,124,68,126,6,6,126,Ø
```

The program redefines the number keys—try pressing any of the numbers from Ø to 9 to see what they look like. To produce effects like this yourself, the first step is to work out where you want to store the DATA for your new characters, as this affects how many of the characters you can define.

There are two standard sets of characters on the Vic: upper and lower case. Each of these can be split into further two parts, normal and reverse characters.

Both the upper and the lower case character sets consist of 256 characters, and each takes 8 bytes of DATA in memory. This means that a whole set of characters takes up 2K of memory, which is rather a lot when you only have 3½K RAM to start off with, as on the unexpanded Vic.

Luckily, you do not need to define a whole character set; you can just define a part of it, although you normally loose the other characters. (There is one special case where you don't lose all the other characters; it is described below.) Of course, you can redefine a complete set of characters if you want to—unless you have an unexpanded Vic, which does not have enough memory for both sets.

There are several possible places where you can store your character set. If you have an unexpanded Vic the possible addresses are 4Ø96, 512Ø, 6144 and 7168. The first of the set is usually the start of BASIC, and so you should not use this. If you want to have a whole redefined character set, you have to use address 512Ø on the unexpanded machine, as the set takes up 2K of memory.

A usual set to use is address 7168, which gives up to 64 user-definable characters, and leaves a reasonable amount of memory left for your BASIC programs. Another advantage of this address is that reverse characters give normal characters—A, B, C, and so on, so you can have user defined characters and a standard alphabet.

The POKEs which you use to redirect the character set pointer to these addresses are as follows:

```
4Ø96:  POKE 36869, 252
512Ø:  POKE 36869, 253
6144:  POKE 36869, 254
7168:  POKE 36869, 255
```

If you type in any of these POKEs, and hit RETURN, you can see the characters on the screen turn into garbage: there are random numbers stored in every address that is not used in RAM, and so the characters you see are composed of these random bytes.

If you hit RUN/STOP and RESET, the characters will be restored to normal. Whenever these two keys are pressed, the character set pointer is always put back to its standard value.

Now that you know how to tell the Vic where to look for your own characters, you can go on to put your DATA in memory and protect it from being overwritten.

The article on pages 38 to 45 explains how to work out the numbers for your characters. Once you have these, you simply POKE them into the memory above BASIC.

After this, it is wise to protect them by moving BASIC's highest possible address down to below the start address of your characters in memory. For example, to protect the memory above 7168, first split the number minus 1 into two parts by dividing by 256. This gives 27 with remainder 255, and these are the numbers you have to POKE into locations 51, 52, 55 and 56 as follows:

```
POKE 51,255: POKE 52,27: POKE 55,255:
POKE 56,27:CLR
```

The Acorn computers have space for up to 32 user-defined graphics characters when first turned on. While this may be enough for normal use, there are numerous occasions when you might want more.

There is a very useful set of commands in BBC BASIC—the *FX commands. They all control some sort of special effect, and there is one—*FX 2Ø—that literally explodes the memory for user definable characters, letting you create more than 32 and letting you redefine characters from the keyboard.

This particular special effects call is only available on machines with either 1.Ø or 1.2 operating systems. Unfortunately, if you have operating system Ø.1 you cannot use this call. If you do not know which operating system your computer has, enter * HELP

A NEW CHARACTER SET

Here is a program which illustrates just one of the possible uses of the *FX command, by redefining some of the keyboard characters, the numbers Ø to 9. First type in this line

```
PAGE=PAGE+&4ØØ
```

and hit RETURN followed by NEW and RETURN. Now type in and RUN the program:

```
1Ø MODE 1
2Ø *FX2Ø,1
1ØØ REM Ø
11Ø VDU 23,48,Ø,124,76,84,86,1Ø2,126,Ø
12Ø REM 1
13Ø VDU 23,49,Ø,8,8,8,24,24,24,Ø
14Ø REM 2
15Ø VDU 23,5Ø,Ø,126,2,126,96,96,126,Ø
16Ø REM 3
17Ø VDU 23,51,Ø,124,4,126,6,6,126,Ø
18Ø REM 4
19Ø VDU 23,52,Ø,96,98,98,126,2,2,Ø
2ØØ REM 5
21Ø VDU 23,53,Ø,124,64,126,6,6,126,Ø
22Ø REM 6
23Ø VDU 23,54,Ø,62,32,126,98,98,126,Ø
24Ø REM 7
25Ø VDU 23,55,Ø,124,4,4,6,6,6,Ø
26Ø REM 8
27Ø VDU 23,56,Ø,6Ø,36,6Ø,1Ø2,1Ø2,126,Ø
28Ø REM 9
29Ø VDU 23,57,Ø,124,68,126,6,6,126,Ø
```

Try pressing any of the number keys to see what they look like.

Although some editions of the BBC User Guide don't say so, the *FX, 2Ø call has seven possible forms; the difference between each is the last number, which can be anything between Ø and 6.

When you turn the computer on the call is set at Ø, and the character definitions are said to be *imploded*. What this means is that you can normally only redefine 32 characters. After a *FX 2Ø,A where A is a number between one and six, the number of characters that you can redefine is increased by A blocks of 32 characters.

This means that the maximum number of characters you can define is 6*32 plus the original set of 32, or 224 characters in all. Even though the computer has ASCII codes for this number of defined characters, it only leaves aside enough memory for 32. So, if you are going to use the *FX command to expand the number of UDGs available, you need to alter PAGE.

PAGE is a variable which contains the address of the start of the BASIC area. So, by changing PAGE you move BASIC's position in RAM. To change it, you simply type

```
PAGE=X
```

where X is the new address of PAGE.

For most programs, you can simply set PAGE to PAGE+&6ØØ. The &6ØØ is a hexadecimal number equivalent to 1536 in decimal. So this leaves you 1536 bytes free to redefine all 192 characters. Sometimes, though, you might want to use some of this memory for a very long BASIC program, and you should set PAGE to a lower address.

Every time that you increase PAGE by &1ØØ the start address of BASIC moves up by 256. This is very convenient, since each block of UDGs also takes up 256 bytes. So, if you use a *FX command, you simply need to press BREAK and then set PAGE like this:

```
PAGE=PAGE+A*&1ØØ
```

The '&' tells the computer that the number is a hexadecimal number—the article on pages 156 to 160 explains what hexadecimal is. It is much easier to use hexadecimal here, since it enables you to add 'A* &1ØØ' to the value of PAGE, instead of 'A*256', which is the alternative.

For example, to use just two extra blocks of UDGs, press BREAK then enter:

```
*FX 2Ø,2
PAGE=PAGE+&2ØØ
```

Once you have reserved memory for the extra graphics characters in this way, you can go on to define and enter the DATA for your new UDGs. You do this in the same way as you would for normal UDGs (see pages 38 to 45).

The Dragon and Tandy have two commands—GET and PUT—which allow you to create and control your own user-defined characters. The article on pages 38 to 45 and pages 350 to 352 explains how to use them.

Since the computers store the DATA for each graphic character in an array, the only limit on how many UDGs you can have is the maximum number of arrays. And, as you probably know, the Dragon and Tandy have just one limit on the number of arrays you can have: the size of the memory.

This means that you can have as many UDGs as you can fit into 32K, if you want to.

Unlike the other computers, the Dragon and Tandy do not allow their users to redefine the ROM characters—the keyboard characters, ROM graphics and so on. While you cannot therefore change the characters that appear in listings and the computer's own messages, you can RUN a program which lets you type with a set of UDG characters.

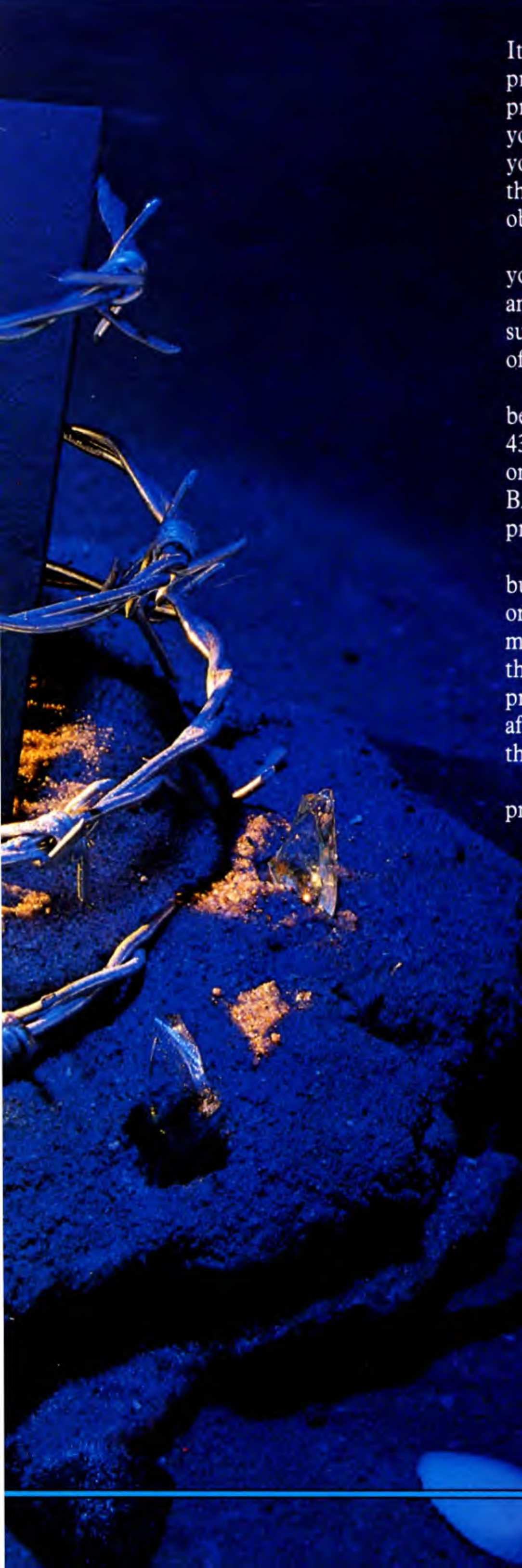
Such a program in BASIC would consist of one 'IF INKEY$="X" THEN . . .' line for every redefined letter, and would be very, very slow if anything like a proper character set was used. You might like to try it for just the number characters, though, using PMODE1 and the DATA in the Spectrum program.

PROTECT YOUR PROGRAMS

Want to protect your special techniques from prying eyes? Or simply add some professional touches to your program? Here's what you can do in BASIC



- WHAT YOU CAN AND CANNOT DO TO PROTECT BASIC PROGRAMS
- BOOTSTRAPS—WHAT THEY ARE—WHAT THEY CAN DO
- PROGRAM INTERDEPENDENCY
- MAKING A PROGRAM AUTORUN
- DISABLING THE NORMAL SAVE AND LIST COMMANDS
- TRICKS TO TRY ON YOUR OWN COMPUTER



It's always a good thing to give a program a professional look once the nuts and bolts programming has been done. At this stage, you can improve program presentation—and you can also provide reasonable protection so that your special techniques remain at least obscured from public inspection.

Some degree of finishing off is essential if you're thinking of marketing your program in any way—particularly if your efforts are to be successful in negotiating the tricky first stages of acceptance by a publishing house.

Cosmetic improvements to displays have been discussed elsewhere (see pages 433 to 439 for example) so here we can concentrate on giving your work some protection. Even a BASIC program can benefit—and it's programs of this level that we'll look at first.

Protection of BASIC programs relies on built-in deterrents—programs written wholly or mostly in machine code can make use of far more sophisticated protection methods. But the techniques employed for BASIC programs (and these need not be simple affairs in spite of the implication) can nevertheless be applied to machine code programs.

All sorts of tricks can be employed to prevent copying by 'pirates', or LISTing by the curious. How many of these you decide to incorporate within your own programs is, of course, up to you. On some programs it simply isn't worth the bother.

The one thing you can be absolutely certain about is that there is no way of protecting a program which makes it completely safe from copying. Many people simply regard program protection as yet another of life's challenges. Others will attempt to break into a program to examine, learn or simply modify a routine for their own purposes.

FIRST STEPS

One method of protecting computer programs is to provide a lot of simple traps. These won't defeat someone who has some knowledge of the machine but would prove a laborious task to evade. Unfortunately, something like this can also be rather tedious for the program writer who has to worry about protection while writing the program. And the program would be very difficult to debug once the protection methods were in place and possibly active.

Simple locks of this type do little more than introduce changes which make it impossible for the normal SAVE, LIST and other editing commands to work.

Their one advantage is that a program has usually got to be RUN before they become active—hence the problems of debugging mentioned previously.

BOOTSTRAP

Somewhat better protection is therefore provided by getting a program to autoRUN as soon as it has LOADed. Now, with the Spectrum and Acorns, this is done simply enough by using suitable LOAD and SAVE commands as you will see later. With the Commodores, pressing a SHIFTed RUN/STOP will autoRUN the first program on tape.

These commands are usually entered in direct mode in the normal process of LOADing a program. But they can just as easily be called up by a separate program LOADed before the main program. Such a program is called a bootstrap and it can be written in BASIC or in machine code depending on the specific tasks it has to carry out. At its very simplest it can take the form:

```
10 LOAD "NEXT PROGRAM'S NAME HERE"
```

RUNning this one-liner would LOAD the program whose name was stipulated.

Q+A

Under what circumstances and in what areas of memory is it possible for me to store routines or calls specifically to implement protection routines?

Each machine allows you to use an area of memory reserved for another device—so long as this is not present.

An unexpected and 'safe' area on the Spectrum is the printer buffer. This is located at 23296 to 23551. This gives you 256 bytes of memory, ample for many routines. You cannot of course use the printer buffer when the program in main memory itself has to access the printer.

The most obvious place to locate machine code routines is in the 'hidden' area of RAM located at 49152–53247 but this is a commonly used location for all sorts of commercial routines.

However there are a number of little nooks and crannies, including free Zero Page space at 251–255, which could be used as a jump location. A slightly larger area occurs at 679–767 (as used in the bootstrap program—see main text). There are also a number of locations both before and after the tape I/O buffer which, too, may be used if the program need not access this device.

'Spare' areas of memory are restricted to Page 9 and 1Ø, also 11 if you're not using the function keys, 12 if you're not using UDGs, and 13 if you're not using disk or Econet systems which access the NMI routines.

The tape buffer at 3ØØ–3FF is the one readily accessible location which may be used. There are a few other locations (all in hex) not used by the Dragon 32; 76–77, E6–77, E6–FF, 114 and 11A–11F

Additional space can be created by clearing more memory for graphics than you actually need. Occasionally, additional RAM may become available when a cartridge is fitted to the computer.

Obviously, a single line such as this is pointless. In reality, bootstraps can be used to do much, much more. They are frequently used to carry supplementary programming, responsible for things like the title page screen displays, copyright notices, loading and playing instructions, setting up variables, and any number of devious protection tricks!

These include some of the most powerful ways of protecting a BASIC program, achieved by playing about with the system commands themselves. But the major use of bootstraps here is as a 'loader' or 'starter' program which allows machine code to be RUN from BASIC.

Bootstraps are used in many types of commercial program: many of these are recorded in the form of multipart programs where each part is called up and LOADed in turn at the command of one or more bootstraps. Remember that BASIC cannot normally be called in without overwriting the bootstrap LOAD command—or, in other words, the bootstrap program itself. So if you're ever tempted to use this technique your self, make sure the machine code modules are called in and LOADed first.

With multipart programs, the protection methods may even rely on a certain level of interdependence between one file (program part) and another. What happens is that one file checks a location value set up by another. Or you could get a program to check a special data file recorded after the main program. In either case, anything missing causes a system crash or program RUN failure.

In most instances, a multipart program will contain one or more modules wholly in machine code and special techniques for protecting these will be covered later.

One final advantage of bootstraps is that faster LOADing times are possible because machine code, screen data and character data can be deposited directly into memory, instead of using BASIC data statements which can do the job only after RUNning.

All this creates the impression of a professionally finished program which, even if written in BASIC, may RUN without ever seeming to do so.

AUTORUN

So how can a bootstrap be used to autoRUN a subsequent program? Sadly this is not too easy on the Dragon and Tandy models, as special calls have to be made to system routines and this is not possible from BASIC.

But on the Spectrum (and the Acorn, as you'll see later), it's simply a case of using the appropriate command at some point within your bootstrap program:

```
990 LOAD "NEXT PROGRAM'S NAME"
```

But remember that this second program will have to have been SAVEd using the autoRUN command SAVE "NEXT PROGRAM'S NAME" LINE 1—or whatever line number represented the start of the program. If a higher start line was chosen, you could even include some copyright notices or code data which could be PEEKed for a security check within REM lines beforehand.

The only way to get a Commodore program to autoRUN is somehow to put LOAD and RUN instructions into the keyboard buffer to simulate pressing SHIFT and RUN/STOP keys, and then pass control back to BASIC from the bootstrap.

This program does just that by playing around with one of the Commodore system routines, called a *vector*—a pair of bytes which tell the operating system which bit of its own code to use for a particular command. More on this shortly. (The Vic version is for the unexpanded Vic.)

```
1Ø N$="NAME"
2Ø POKE 49189, LEN(N$)
3Ø FOR Z=1 TO LEN(N$)
4Ø POKE 49189+Z, ASC
   (MID$(N$,Z,1))
5Ø NEXT Z
6Ø FOR Z=679 TO 736
7Ø READ X
8Ø POKE Z,X
9Ø NEXT Z
1ØØ FOR Z=49152 TO 49188
11Ø READ X
12Ø POKE Z,X
13Ø NEXT Z
2ØØ POKE 77Ø,167: POKE 771,2:
   SYS49152
21Ø PRINT "OKAY": GOTO21Ø
22Ø DATA 169,47,133,Ø,169,55,133,1,32,
   138,255,169,1,141,32,2Ø8
23Ø DATA 169,48,141,119,2,169,76,141,
   12Ø,2,169,2Ø7,141,121,2
24Ø DATA 169,13,141,122,2,169,82,141,
   123,2,169,213,141,124,2
25Ø DATA 169,13,141,125,2,169,7,133,
   198,1Ø8,Ø,16Ø
26Ø DATA 162,1,16Ø,1,169,1,32,186,255,
   162,38,16Ø,192,173,37,192
27Ø DATA 32,189,255,169,167,133, 251,
   169,2,133,252,162,5,16Ø,3,
   169,251
28Ø DATA 32,216,255,96
```

```
1Ø N$="NAME"
2Ø POKE 72Ø5, LEN(N$)
3Ø FOR Z=1 TO LEN(N$)
4Ø POKE 72Ø5+Z, ASC
   (MID$(N$,Z,1))
5Ø NEXT Z
6Ø FOR Z=679 TO 723
7Ø READ X
8Ø POKE Z,X
9Ø NEXT Z
1ØØ FOR Z=7168 TO 72Ø4
11Ø READ X
12Ø POKE Z,X
13Ø NEXT Z
2ØØ POKE 77Ø,167: POKE 771,2: SYS7168
21Ø PRINT "OKAY": GOTO21Ø
22Ø DATA 32,135,255
23Ø DATA 169,48,141,119,2,169,76,141,
    12Ø,2,169,2Ø7,141,121,2
24Ø DATA 169,13,141,122,2,169,82,141,
    123,2,169,213,141,124,2
25Ø DATA 169,13,141,125,2,169,7,133,
    198,1Ø8,Ø,192
26Ø DATA 162,1,16Ø,1,169,1,32,186,255,
    162,38,16Ø,28,173,37,28
27Ø DATA 32,189,255,169,167,133,251,
    169,2,133,252,162,5,16Ø,3,169,251
28Ø DATA 32,216,255,96
```

SAVE this program so you can call it up for use when required. When you do, reLOAD it and enter the name of the program you wish to autoRUN in Line 1Ø. This second program must subsequently be SAVEd after the bootstrap if you're using tape. RUN the bootstrap program to place it in memory. Position your tape and then SAVE the bootstrap, which should now carry the name of the program to be autoRUN. When the OKAY message is displayed press the RUN/STOP and RESTORE keys to break out of the program. LOAD in the second program, and PEEK locations 45 and 46 (locations for the start of variables). Now enter the following line at the start of the second program:

```
Ø POKE 45, X : POKE 46, Y
```

Where X and Y are the values you've just obtained by PEEKing those two locations. RETURN the line and immediately repeat the PEEK, in direct mode, to see if the values have changed. Amend line Ø to the new figures. Repeat the cycle again until the value remains constant.

The second program can now be SAVEd. But first you may wish to incorporate other security checks, such as disabling POKEs (see page 379).

The Acorn machines have a specific command for LOADing and RUNning, and it is used within a bootstrap in the form:

```
99Ø CHAIN "NEXT PROGRAM NAME"
```

And that's all there is to it! As suggested by the command, other programs may be CHAINed by an appropriate program line within the first program (and this is in fact how the 'Welcome' tape goes from one program to the next).

USING SYSTEM VARIABLES

But autoRUNning is not enough on its own. You also have to provide some means of ensuring that the program cannot be stopped, LISTed, or amended.

The way to do this is to make adjustments to the way the system reacts when a particular call is made to, for instance, the LIST subroutine.

Every computer comes complete with an operating system and, as far as we are concerned, a BASIC interpreter. Most of the system information is held in read only memory, ROM, but some of this is transferred to random access memory, RAM, when the computer is switched on. And it is this information that can be changed by the programmer to alter the way the system operates—which enables pretty sophisticated security measures to be incorporated within programs as we'll see in later articles.

As you are going to be playing around with the workings of the computer, you will need a documented list of system variables and subroutines. And if you wish to delve further into the system, a good memory map is also essential. See your reference manual.

When the computer is switched on, some of the system information is down-loaded from ROM into RAM to allow the operating system to alter the value of some of its variables. As this information is in RAM it is vulnerable to any changes the programmer may care to make.

A special type of system variable is the system RAM *pointer* or *vector*. This is normally two adjacent locations which hold the address of a specific system subroutine. If the address is changed, then the operating system is redirected whenever that subroutine is called. All these system subroutines are of course in machine code and that is why protection by adjustment of these is better left to your programs which are also in machine code.

Those pointers or vectors of particular interest as far as we are concerned include the equivalents of the LIST vector, SAVE vector, INTERRUPT vectors and the WARM START or RESET vector. The last two types are really beyond the scope of this article and the exact vectors used vary on different machines.

CASE STUDIES

It is difficult if not impossible to provide complete protection from piracy—especially if you are trying to avoid getting heavily involved in machine code. The methods you can use on any particular machine vary immensely because the operating systems differ so much. But have a look at some of the following ideas:



One of the simplest checks is to insert an untouchable copyright statement within your program. This could even be linked to a routine which PEEKs to check its presence, doing a system reset if it has been tampered with in any way.

First find the address of the BASIC program area. This address (the system variable PROG) is held at locations 23635 and 23636, and can be determined by PRINTing PEEK 23635 + 256 * PEEK 23636. Once you know the value of PROG, you can poke any number N into (PROG + 1) and the first line of your program will change to Line N.

The secret is to make N equal to zero because it is not possible to get rid of Line Ø in a program. Suppose the first line is:

```
10 REM (c) BLOGGS 1984
```

Enter the direct command:

```
POKE (PEEK 23635+256*PEEK 23636)+1,0
```

And you're there: Line 1Ø then becomes Line Ø. Obviously this could be done from within a bootstrap program.

As far as an autoRUN program is concerned, the obvious way of stopping this is to press BREAK. This puts a BREAK message in the lower part of the screen and then allows the program to be LISTed.

But suppose the lower part of the screen will not accept the message? If you look at the list of system variables given in the Spectrum handbook, you can see that DF SZ (at location 23659) holds the number of lines in the lower part of the screen (the figure is normally 2). If you POKE Ø into 23659, the computer will crash as soon as a message tries to appear in the lower part of the screen.

You cannot enter this as a direct command but must include it in a program. For example, try this short demonstration:

```
10 POKE 23659, 0
20 PRINT AT 5,5; RND
30 GOTO 20
```

And then RUN the program. Pressing the BREAK key causes an inescapable crash.

If you use this method, be careful that your program does not actually need to display messages such as INPUT? or scroll? or it will crash as well. Therefore, use INKEY$ for input sequences.

But from the pirate's point of view the favourite way of preventing a program from autoRUNning is to MERGE it rather than LOAD it. Again the handbook gives you a clue how to get round this hiccup: you cannot MERGE 'bytes'. So if you SAVE your program as CODE (in other words, as 'bytes'), the would-be pirate is foiled. You have to SAVE everything—all the system variables, all the BASIC variables, the program and all the spare memory, the lot—above the printer buffer.

Thus the CODE at which SAVEing starts is 23552, which is the start of the system variables. The number of bytes is N − 23552 where N is any large number greater than STKEND (the address of the start of spare space). The handbook gives this address as PEEK 23653 + 256*PEEK 23654. If you want to SAVE all of the user RAM memory, then make N equal to 65535. This is the maximum number of bytes you can save to tape. So put the following line at the start of your program:

```
1 SAVE "PROGNAME" CODE 23552,
   N-23552
```

Then add a second line with the POKE to make the program crash on BREAK:

```
2 POKE 23659, 0
```

Now type GOTO 1 and so SAVE the program. The command LOAD "PROGRAM NAME" CODE will cause the program to autoRUN.

The ordinary LOAD command will not work if N is very large—for example, 65ØØØ for the 48K machine. It is wasteful on tape and memory and as there is hardly sufficient room in the computer's memory for your own program, nothing else—such as a copy tape program—will fit, so your program is safe.

The autoRUN program and any keyboard disabling are easily recognisable and a further ploy is needed to augment these. This is to use the back-delete technique to conceal the presence of POKEs used to alter memory and disable certain keys. In fact, the technique may be used to disguise anything in a program line—even data used for a security check!

The back delete technique uses embedded delete commands contained within quotes following the non-active REM part of a program line. An example shows clearly how this technique works, so type in the following—with no spaces:

```
99 POKE45,0:POKE46,20:RUN:
   REM""SYS4096
```

Now move the cursor to the second set of quote marks and press SHIFT and the INST/DEL key to insert 27 spaces. As soon as this has been done press the unSHIFTed INST/DEL key to enter 27 deletes. These symbols take the form of reverse Ts. Press RETURN to enter the line. Again move the cursor up to edit it, but this time delete the last set of quote marks. Enter the line by pressing RETURN. Now try LISTing it. If you've followed the steps correctly all that should be displayed is:

```
10 SYS4096
```

Try editing the series of embedded deletes by inserting extra spaces and adding further delete symbols.

You can in fact wipe an entire line using this technique. Or you can choose to wipe out only part of a line in a program.

If these program lines appear at the start and end of a screen listing, then no obvious gaps will be left but the momentary flash of the disguised line may give the game away. And of course this technique applies only to the screen display: a listing taken from a printer will reveal all!

However, the very presence of a SYS call may be enough to deter an intruder afraid that a machine-code program was looming.

If you are using a bootstrap program, another method of providing additional protection to the autoRUN program is to incorporate an additional program line within the autoRUN program which PEEKs locations used by the bootstrap. This would mean that the autoRUN program could not be LOADed and RUN other than by the bootstrap. If you've used the bootstrap program earlier, add the

Q+A

How do I load a machine code routine from BASIC so that it remains transparent to the user?

You can use a bootstrap as below. N is the start location of the machine code routine in hex on the Acorns.

```
10 CLEAR N-1
20 LOAD "m/c file" CODE
```

SAVE this program, using SAVE "loader name" LINE 1Ø so it autoRUNs. Then after it on the tape, SAVE the machine code routine (SAVEd using SAVE "m/c file" CODE N,B where B is the number of bytes required).

```
* LOAD""N
```

```
10 CLEAR 200,N-1
20 CLOADM "M/C FILE"
```

following line to your second program:

```
1 IF PEEK (679) < > 169 OR PEEK (68Ø)
   < > 47 THEN NEW
```

```
1 IF PEEK (679) < > 32 OR PEEK (68Ø)
   < > 135 THEN NEW
```

Also include suitable keyboard disable POKEs so that the RUNning program cannot be interrupted when under way.

The only way to make your programs really secure is to write a short section of machine code. The problem with using a BASIC program is that whatever you do relies on the program being RUN—whether it is resetting variables, altering memory or making the program crash when BREAK is pressed. All the pirate has to do is LOAD the program and then LIST it before it is RUN—thus displaying all your clever tricks. This is why a short bootstrap or 'header' program in machine code is so useful—if the pirate tried to LOAD and LIST that, all that would result would be a 'Bad program' message.

However there is one sure-fire protection method that will foil all but the real machine code experts and, surprisingly, it doesn't actually use any machine code itself.

What it does is alter the very last byte of the program. This is always &FF (in hex) for a BASIC program. If you alter this byte and then try to LIST the program, the computer is unable to find the end of the program and so gives the inevitable 'Bad program' message. But you must make sure the program ends properly so add END as the last line to be safe. Here's what to do:

First type in this line:

```
PRINT ~PAGE, ~TOP
```

(Don't worry that the ~ sign appears as ÷ in MODE 7 on the BBC as they mean the same thing—all it does is to convert a decimal number into the more convenient hex.) You'll see two numbers printed on the screen. These are PAGE, where your program starts in memory, and TOP, where it ends. PAGE is usually ØEØØ (or 19ØØ if you have an Acorn disk filing system). TOP depends on the size of your program. Now enter this:

```
?(TOP-1)=Ø
```

This sets the last byte of your program to zero. Now all you have to do is save the program using the *SAVE command:

```
*SAVE "program name"□MMMM□NNNN
```

Where MMMM and NNNN are the two numbers for PAGE and TOP you found earlier. Your program is now safe.

Anyone using the program must load it using LOAD "progname", as the CHAIN command will not work. It will RUN normally but it is impossible to LIST.

To get a BASIC program to autoRUN after LOADing from tape requires some fairly tricky machine code routines when SAVEing to tape.

A simpler method to protect your BASIC program is to SAVE it as 'machine code', using the command CSAVEM, and to include a short machine code routine within that program to reset the BASIC pointers. You then RUN the program without returning to the normal direct command mode. Here's how to do it.

The BASIC pointers which need to be reset are the start address of the program, which is given by:

```
PEEK(25)*256+PEEK(26)
```

Similarly you can obtain the end address with:

```
PEEK(27)*256+PEEK(28)
```

Another important pointer which prevents LISTing of the program is contained in the first two bytes of the BASIC program. If both these bytes are set to zero, the Dragon will not be able to LIST or RUN the program.

To add this protection to your BASIC program, first debug your program. Then RENUMber it so that its lowest line number is 1Ø. Then add the following lines. Be especially careful to type in Line 1 exactly as shown, including the all-important space:

```
1 REM□AAAAAAAAAAAAAAAAAAAAAAAAAAA
2 ST=PEEK(25)*256+PEEK(26):
   A$="3Ø8C1ØEC81DD19EC81ED
   8CEEEC84DD1B7E85A5"
3 FOR K=1 TO 38 STEP 2: POKE ST+6+
   K/2, VAL("&H"+MID$(A$,K,2)): NEXT
4 POKE ST+25, PEEK(25): POKE ST+26,
   PEEK(26): POKE ST+27,PEEK(ST): POKE
   ST+28, PEEK(ST+1): END
```

At this point make sure you CSAVE the program normally. When you RUN this program, a short machine code routine is POKEd into the REM statement of Line 1 by Lines 2 and 3. Information for the BASIC start pointer is also POKEd into the REM statement by Line 4. RUN the program and then try to LIST Line 1. You should see that it has changed.

To prevent the final program being broken into, it is necessary also to disable the BREAK key and the more powerful RESET button. To do this, enter the following line after deleting Lines 3 and 4 of the previous program. These have already done their job:

```
2 A$="E4EDØ4CBE4EC": FOR K=1
   TO 12 STEP 2: POKE 416-K/2,VAL("&H"
   +MID$(A$,K,2)): NEXT: POKE113,Ø
```

The FOR ... NEXT loop, when RUN, prevents BASIC looking at the BREAK key, and the last POKE will cause the program to be NEWed if the RESET button is pressed. However, this is not suitable for programs with INPUT lines.

Before you SAVE the protected program you have to add information about the end of the BASIC program to the REM statement of Line 1. You can't do this in program mode, as this information immediately changes value when any line number is ENTERed. Therefore, in direct mode enter the following:

```
ST=PEEK(25)*256+PEEK(26): POKE
ST+29, PEEK(27): POKE ST+3Ø, PEEK(28):
   POKE ST,Ø: POKE ST+1,Ø
```

Press ENTER. Then, to SAVE the protected program use:

```
CSAVEM "PROGRAM NAME",ST,
   PEEK(27)*256+PEEK(28),ST+6
```

The program is SAVEd in its protected form ready for use at any time in the future.

To LOAD the program from tape use CLOADM instead of CLOAD. After LOADing, the Dragon will be unable to find a BASIC program so LIST and RUN will have no effect. To RUN the program you have to use the command EXEC followed by ENTER.

Microtip

The Commodores can enter several machine code routines using a bootstrap. But precautions have to be taken as the computer returns to the beginning of the program and so will attempt to reLOAD the first file each time. To prevent this, "flag" a variable at the start of the program:

```
1Ø IF A=Ø THEN A=1:LOAD"FILE1"1,1
2Ø IF A=1 THEN A=2:LOAD"FILE2",1,1
```

When the program returns to the beginning of the program, it will note that it has already given variable A (the flag) a value, and go straight away to Line 2Ø to LOAD the next file.

PROGRAMMING FOR JOYSTICKS

Joysticks are the key to more professional games. But making your games more enjoyable doesn't mean that you have to learn machine code—you can start with BASIC

One obvious difference between commercial games and home-produced ones is often the provision of a joystick option. You don't have to zoom off into the heady realms of machine code to use joysticks with your own programs, though.

This time in Games Programming you'll see how to use joysticks with programs written in BASIC, making your games programs look more professional and more fun to play.

In the next part of Games Programming the joystick routine will be used in a game, so don't forget to SAVE the program.

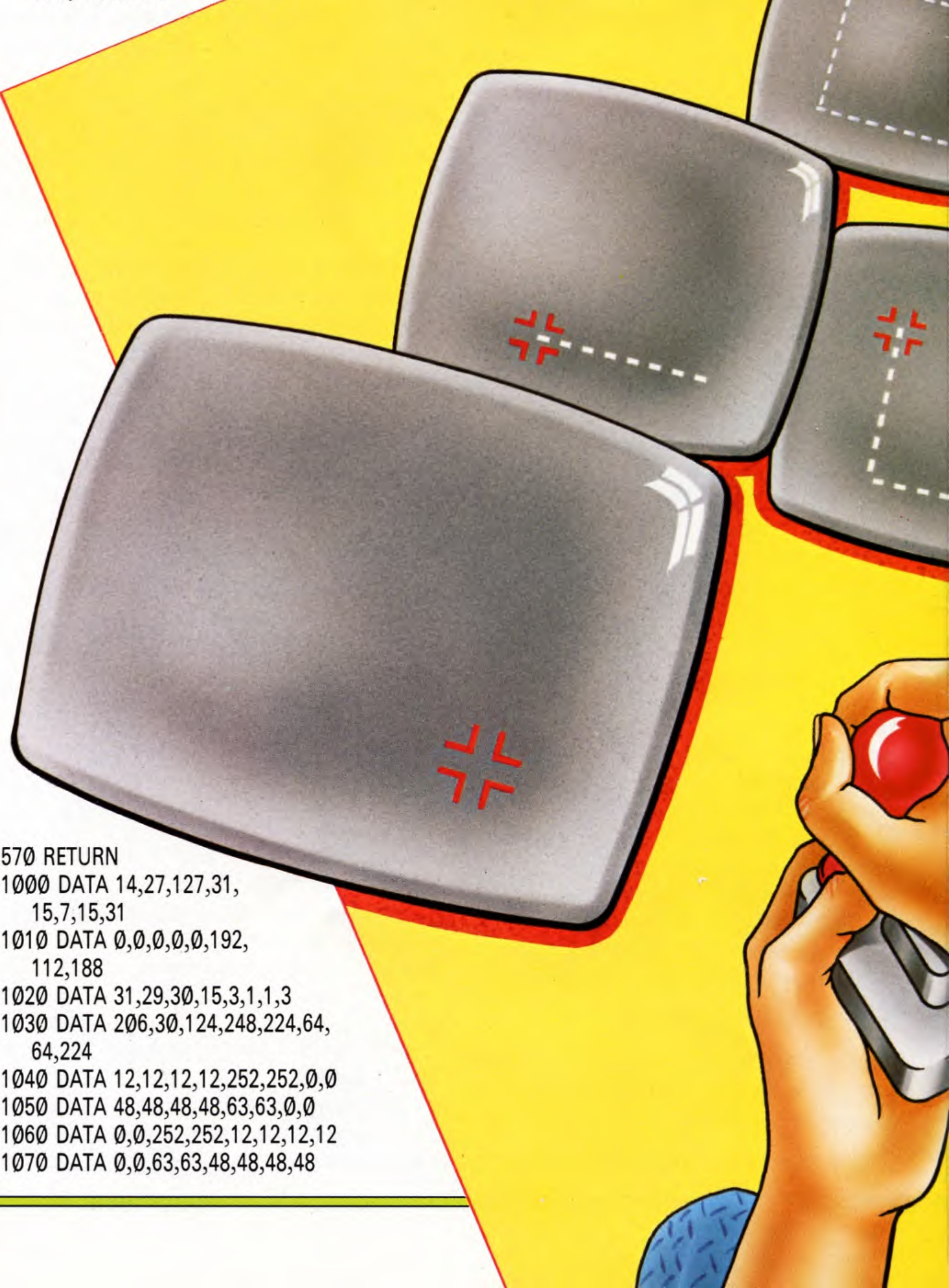
Before you can use a joystick program, you will need a suitable joystick. Each computer's program is written to suit particular joystick standards for that machine, and you'll find notes on this at the beginning of the programming. And there is more information on joystick hardware in general in the article on pages 220 to 224.

There is a very large number of joysticks available for the Spectrum. As they all work in a slightly different way it is not possible to write routines for use with all those available. Instead, you'll see how to write a routine which will read by the most widely-used type, the Kempston joystick, and those which use the Kempston standard.

A GUNSIGHT

Type in this first section of program and you'll see a gunsight which you can control with your joystick:

```
100 BORDER 1: PAPER 1: INK 7: OVER 1: CLS
    : INK 8
110 FOR n=USR "a" TO USR "h"+7: READ
    a: POKE n,a: NEXT n
130 LET s=0: LET x=15: LET y=10
140 PRINT OVER 1;AT y,x;CHR$ 148;CHR$
    149;AT y+1,x;CHR$ 150;CHR$ 151
200 GOSUB 500
480 GOTO 200
500 IF IN 31=0 THEN RETURN
510 PRINT OVER 1;AT y,x;CHR$ 148;CHR$
    149;AT y+1,x;CHR$ 150;CHR$ 151
520 IF (IN 31=8 OR IN 31=9 OR IN
    31=10) AND y>1 THEN LET y=y-1
530 IF (IN 31=4 OR IN 31=5 OR IN 31=6)
    AND y<20 THEN LET y=y+1
540 IF (IN 31=1 OR IN 31=5 OR IN 31=9)
    AND x<30 THEN LET x=x+1
550 IF (IN 31=2 OR IN 31=6 OR IN
    31=10) AND x>0 THEN LET x=x-1
560 PRINT OVER 1;AT y,x;CHR$ 148;
    CHR$ 149;AT y+1,x;CHR$
    150;CHR$ 151
570 RETURN
1000 DATA 14,27,127,31,
    15,7,15,31
1010 DATA 0,0,0,0,0,192,
    112,188
1020 DATA 31,29,30,15,3,1,1,3
1030 DATA 206,30,124,248,224,64,
    64,224
1040 DATA 12,12,12,12,252,252,0,0
1050 DATA 48,48,48,48,63,63,0,0
1060 DATA 0,0,252,252,12,12,12,12
1070 DATA 0,0,63,63,48,48,48,48
```



- COMPATIBILITY WITH DIFFERENT JOYSTICKS
- HOW TO READ JOYSTICKS WITHIN A BASIC PROGRAM
- ANIMATING A GUNSIGHT
- MAKE YOUR GAMES MORE PROFESSIONAL
- ELECTRON INTERFACE
- TESTING FOR THE FIRE BUTTON



The program begins by initializing the screen colours. Next, eight UDGs are created using the DATA in Lines 1ØØØ to 1Ø7Ø. These are not all used yet, because you are not only defining a gunsight, you are also creating a picture of a duck which will be used when you type in the remainder of the program (covered in the next part of this article).

Having created the UDGs, Line 13Ø sets the start position of the gunsight. The line also sets the score to zero but, again, this won't be needed until later.

Line 14Ø PRINTs the top half of the gunsight using two UDGs, followed by the bottom half using two more. The gunsight is controlled by calling the subroutine starting at Line 5ØØ—the calling is done by Line 2ØØ.

The joystick subroutine, starting at Line 5ØØ looks at the values of IN 31 to detect which way the joystick is being pushed. What the IN function does is to look at a particular port within the machine to see which value is being returned. The Kempston joystick addresses port number 31, so the program looks at the values of IN 31.

The values that IN 31 can take are shown in **fig. 1.** The four main directions give values of 1, 2, 4 and 8. In addition, diagonal joystick movements can be detected by looking at the total of the two adjoining directions—see **fig. 1** again.

The first line in the subroutine—Line 5ØØ—detects if the joystick is positioned centrally. IN 31 will be zero if you leave the joystick alone and it springs back to its central position.

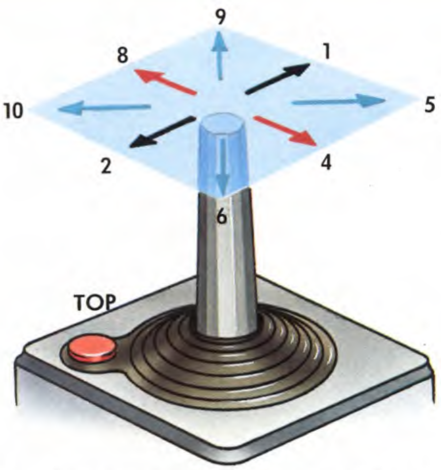
Line 51Ø blanks out the gunsight because this is the second time you've PRINTed OVER 1—the first was in Line 14Ø. OVER 1 blanks out the graphic when it is used for the second time, so the graphic will disappear.

Now that the previous position has been blanked out, a new position can be calculated. The new position depends on which way the joystick has been pushed by the player. Line 52Ø detects the three positions which indicate a downwards movement—straight down and the two downward diagonals. Line 53Ø detects the three upward possibilities, and Lines 54Ø and 55Ø detect the left and right positions. The diagonal position values appear in two lines because a diagonal movement is made up of two up or down, and left or right, values. If you're a little confused, look back at **fig. 1** and you should see what's happening.

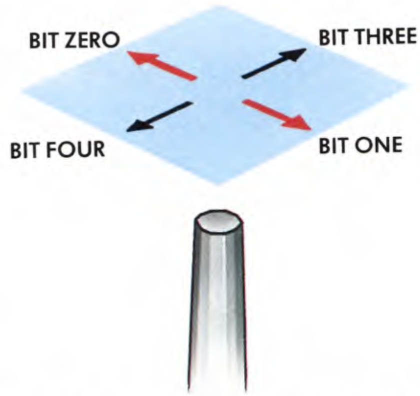
The subroutine ends by PRINTing the gunsight at the new position. Notice that as this is the first time the gunsight has appeared at that position, it appears normally.

To allow you to continue moving the gunsight, Line 48Ø says GOTO 2ØØ as a temporary measure so that the joystick routine can be called repeatedly.

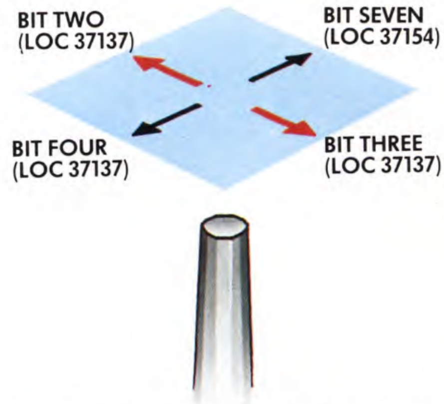
The Commodore 64 program is compatible with any Atari-type joystick. First connect your joystick to the socket marked 1.



1. The Spectrum joystick gives one of these eight values according to which way the joystick is pushed



2. The Commodore 64 joystick sets these bits in location 145. ANDing will tell you the joystick's direction



3. The Vic 20 joystick sets bits in two separate memory locations, according to the direction of the stick

A GUNSIGHT

Now type in this first section of program which will allow you to use your joystick to move a gunsight around the screen.

In the next part of this article, you'll see how to turn the program into a complete duck-shooting game.

```
20 POKE 5328Ø,5:POKE 53281,Ø:
   PRINT "♡"
3Ø Y=1284:TU=Ø:SC=Ø
4Ø FOR Z=1 TO 23:PRINT "π▩
   □□□□□□□□□□□□□□□
   □□□□□□□□□□□□□□□
   □□□□□□□□▩";:NEXT Z
11Ø J=PEEK(145)
12Ø POKE Y,32:POKE Y+1,32:POKE
   Y+4Ø,32:POKE Y+41,32
13Ø T=Ø:T1=Ø:T2=Ø:T3=Ø:T4=Ø:
   IF ((JAND1)=Ø)=-1 THEN
   T1=-4Ø
14Ø IF ((JAND2)=Ø)=-1 THEN T2=4Ø
15Ø IF ((JAND4)=Ø)=-1 THEN
   T3=-1
16Ø IF ((JAND8)=Ø)=-1
   THEN T4=1
165 T=T1+T2+T3+T4
17Ø IF(Y+T>1Ø63ANDY+T<1943)
   AND(PEEK(Y+T)<>1Ø2AND
   PEEK(Y+T+1)<>1Ø2)THEN
   Y=Y+T
18Ø POKE Y,125:POKE Y+1,1Ø9:POKE
   Y+4Ø,11Ø:POKE Y+41,112
19Ø GOTO 11Ø
```

Line 2Ø sets the border and background colours and clears the screen. The gunsight's start position is set by Line 3Ø along with the number of ducks and the score—the variables TU and SC won't be needed until later. The border is drawn by Line 4Ø.

The routine from Lines 11Ø to 18Ø deals with the joystick. PEEKing location 145 can provide you with information about which of the four directions—up, down, left and right—the joystick is being pushed, and whether the fire button is being pressed.

To detect movement of the joystick, the logical operator, AND, is used. For more information about how this works, see the article on pages 284 to 288. Bit zero is set if the stick is pushed up. ANDing it with 1, then, will yield zero if the joystick is pushed in that direction. Similarly, for down, bit one is set, and J needs to be ANDed with 2. For left, J needs to be ANDed with 4, and for right, J needs to be ANDed with 8. As you will see next time, J needs to be ANDed with 16 to detect pushes on the fire button.

In Line 13Ø, T, T1, T2, T3 and T4 are set to zero. This set of variables is used to adjust the gunsight's screen position according to the results from ANDing with J. Lines 13Ø and 14Ø deal with vertical movements of the gunsight, while Lines 15Ø and 16Ø deal with left and right movements.

Line 165 calculates the overall change in screen position. If the gunsight isn't going to move off screen, a new screen position is calculated by Line 17Ø. The POKEs in Line 18Ø display the gunsight on the screen again.

The Vic 20 program is very similar to the program designed for the Commodore 64. As usual, some of the memory locations are different, and the screen dimensions differ.

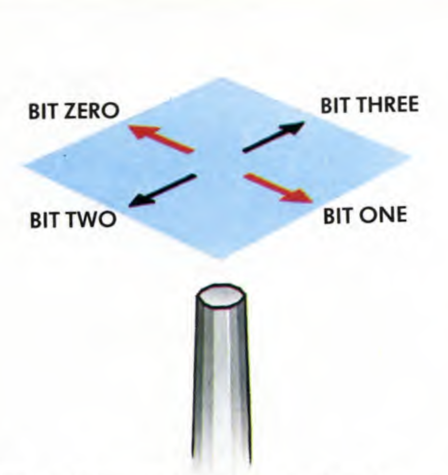
A GUNSIGHT

Type in this program and you will have a gunsight to move around the screen:

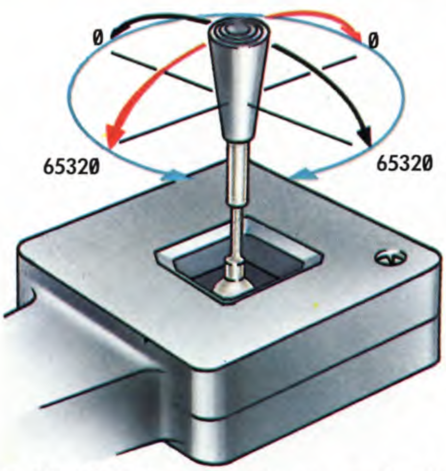
```
4Ø FOR Z=Ø TO 7:POKE 7424+Z,
   Ø:NEXT Z
5Ø POKE 36879,13:PRINT "♡":
   POKE 36869,255
6Ø Y=7791+88*2:POKE 36878,
   15:TU=Ø:SC=Ø
7Ø FOR Z=1 TO 22:POKE 768Ø+
   Z*22,23Ø:POKE 77Ø1+Z*22,23Ø:NEXT Z
15Ø POKE 37139,Ø:POKE 37154,127:
   J=PEEK(37137):JØ=((JAND4)=Ø):
   J1=((JAND8)=Ø)
16Ø J2=((JAND16)=Ø):F=((JAND32)
   =Ø):J3=((PEEK(37152)AND
   128)=Ø):POKE 37154,255
17Ø POKE Y,32:POKE Y+1,32:POKE
   Y+22,32:POKE Y+23,32
18Ø T=Ø:T1=Ø:T2=Ø:T3=Ø:T4=Ø:
   IF JØ=-1 THEN T1=-22
19Ø IF J1=-1 THEN T2=22
2ØØ IF J2=-1 THEN T3=-1
21Ø IF J3=-1 THEN T4=1
22Ø T=T1+T2+T3+T4
23Ø IF (Y+T>77Ø1ANDY+T<8163)
   AND(PEEK(Y+T)<>23ØANDPEEK
   (Y+T+1)<>23Ø)THENY=Y+T
24Ø POKE Y,253:POKE Y+1,237:
   POKE Y+22,238:POKE Y+23,24Ø
25Ø GOTO 15Ø
```

First of all, Line 4Ø defines the space character. Line 5Ø sets the screen colour, clears the screen and sets up the graphics mode. Line 6Ø is another initialization line. It is concerned with setting the start position of the gunsight—Y—and setting up some things that won't be needed until next time: the sound effect volume, the number of ducks—TU—and the score—SC. Line 7Ø sets up the border.

The joystick routine itself runs from Line 15Ø to Line 24Ø. Before you can use a



4. The Electron joystick sets these bits as the joystick is pushed, returning to zero when released



5. The BBC joystick returns a value in the range 0 to 65320 from each of the two potentiometers

joystick, location 37139 must be POKEd with Ø, and location 37154 must be POKEd with 127.

To find our which way the joystick is being pushed, the contents of locations 37137 and 37154 must be examined. In location 37137, bit two is set when the joystick is pushed upwards, bit three when it is pushed downwards, and bit four when it is pushed to the left. When the joystick is pushed to the right, it sets bit seven of location 37154—not very logical at all! If you move the stick diagonally two bits are set.

Lines 15Ø and 16Ø use logical ANDs to find out which way the joystick is being pushed—JØ to J3 correspond to up, down, left and right. In addition, the fire button is checked—bit five of location 37137 is set when the button is pressed. If the bit that is being tested is set, the J variable becomes −1.

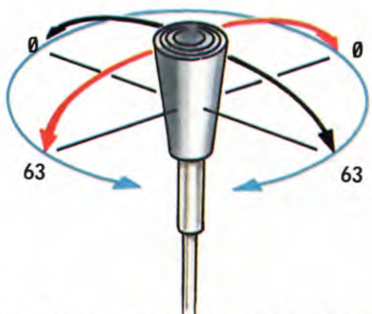
In Line 18Ø T, T1, T2, T3 and T4 are set to zero. This set of variables is used to adjust the gunsight's screen position according to the values of JØ, J1, J2 and J3. Lines 18Ø and 19Ø deal with vertical movements of the gunsight, while Lines 2ØØ and 21Ø deal with left and right movements.

Line 22Ø calculates the overall change in screen position. If the gunsight isn't going to move off screen, a new screen position is calculated by Line 23Ø. The POKEs in Line 24Ø display the gunsight back on the screen.

Line 25Ø is simply a temporary measure so that the joystick can control the gunsight continuously.

This program will only work with the BBC. Electron owners should look at the next one.

If you are using Acorn's own joysticks,



6. The Dragon and Tandy joysticks return a value in the range 0 to 63 from each of the two potentiometers

you'll find that the program only responds to one of the pair of joysticks.

Type in this first section of program and you'll see a small gunsight—a plus sign—which you can move around the screen:

```
20 DIMV(2),V2(2)
25 V(1)=680:V(2)=512
30 *TV255,1
60 MODE1
70 VDU23;8202;0;0;0;
100 GCOL3,3
110 VDU5
115 MOVE 680,512:PRINT"+"
120 PROCUPDATE(1)
130 V(1)=1280-V(1)
140 PROCUPDATE(2)
160 MOVE V2(1),V2(2):PRINT"+"
170 MOVE V(1),V(2):PRINT"+"
220 GOTO 120
230 DEF PROCUPDATE(P)
240 V2(P)=V(P)
250 V(P)=(ADVAL(P))/(53+
    13*((P+1) MOD 2))+20
260 ENDPROC
```

At the start, two arrays are DIMensioned. They will be used to store the current position and the last position of the gunsight.

Line 3Ø prepares the screen for the game, while Line 6Ø selects MODE1. The text cursor is switched off by Line 7Ø.

In order that the gunsight can be blanked out in the course of animating it, Line 1ØØ sets up an exclusive OR—see page 372—on white. The effect of this is that printing a white graphic on top of another white graphic will cause the areas where they overlap to disappear. The VDU5 in Line 11Ø allows you to PRINT at the graphics cursor and use GCOL to colour the text characters.

PROCUPDATE deals with the input from the joystick. The Acorn joysticks come as a pair which plug into the Analogue to Digital (A to D) converter at the rear of the machine. The ADVAL function returns a value that is being sent through the A to D converter. In the case of joysticks, ADVAL(1) reads a value for the horizontal position of one joystick, and ADVAL(2) reads the vertical position. ADVAL(3) and ADVAL(4) read the horizontal and vertical position of a second joystick. Each of the four ADVALs can return a value from Ø to 6532Ø in steps of 16.

PROCUPDATE will work for either ADVAL(1) or ADVAL(2). The value of P is passed from the PROCedure calls in Lines 12Ø and 14Ø. Line 24Ø swaps either the vertical or the horizontal element of V2 for the corresponding element in V. This is swapping the last position for the current position, ready for calculating a new position and subsequently moving the gunsight. Line 25Ø's job is to calculate the screen coordinate which corresponds to either the horizontal or vertical position of the joystick. When dealing with ADVAL(1), Line 13Ø adjusts the newly-calculated value of V(1).

Now that both V(1) and V(2) have been calculated, the gunsight can be blanked out and replotted at its new position. Line 16Ø sees to the blanking out—remember that you are using an exclusive OR, so that plotting a second time will make the gunsight disappear. The gunsight is replotted by Line 17Ø.

In order to use the joystick to move the gunsight continuously rather than just one step every time the program is RUN, you will need to close a loop—this is the function of Line 22Ø, which is a temporary measure only.

JOYSTICKS ON THE ELECTRON

There is no direct way of attaching joysticks to the Electron, as there is no built-in joystick socket. This doesn't mean that you can't use joysticks with the Electron—but you do need to buy a separate interface. There are a

number of suitable ones on the market, but this program has been written for use with the First Byte Joystick Interface which allows you to connect any Atari-type joystick.

```
5 DV=32
20 DIMV(2),V2(2)
25 V(1)=680:V(2)=512
30 *TV255,1
60 MODE1
70 VDU23;8202;0;0;0;
100 GCOL3,3
110 VDU5
115 MOVE 680,512: PRINT"+"
120 PROCUPDATE
160 MOVE V2(1),V2(2):PRINT"+"
170 MOVE V(1),V(2):PRINT"+"
220 GOTO 120
230 DEF PROCUPDATE
232 V2(1)=V(1):V2(2)=V(2)
240 V=255-?&FCC0
242 IF V AND 1 THEN V(2)=
    V(2)+DV
244 IF V AND 2 THEN V(2)=
    V(2)-DV
246 IF V AND 4 THEN V(1)=
    V(1)-DV
248 IF V AND 8 THEN V(1)=
    V(1)+DV
250 IF V(1)<0 OR V(1)>1240 THEN
    V(1)=V2(1)
255 IF V(2)<0 OR V(2)>1000 THEN
    V(2)=V2(2)
260 ENDPROC
```

Line 5 sets DV at 32. DV will be used to move the gunsight in steps of 32 pixels. Line 20 DIMensions two arrays which will be used to store the gunsight's last position and current position. The initial position of the gunsight is set by putting 680 and 512 into array V.

Line 30 prepares the screen for the game, whilst Line 60 selects MODE1. The text cursor is switched off by Line 70.

In order that the gunsight can be blanked out in the course of animation, Line 100 sets up an exclusive OR—see page 372—on white. The effect of this is that printing a white graphic on top of another white graphic will cause the areas where they overlap to disappear. Next, the VDU5 in Line 110 allows you to PRINT at the graphics cursor and use GCOL to colour the text.

PROCUPDATE deals with the input from the joystick. Line 232 swaps the values in the current position array—V—into the last position array—V2. Line 240 takes the value from the port being addressed by the joystick. It's subtracted from 255 to make the testing in Lines 242 to 248 easier.

The checks look for bits set by the signals from the joystick. Line 242 tests bit zero, for when the joystick is pushed up, and Line 243 tests bit one, for when the joystick is pushed down. Similarly, Lines 246 and 248 check bit two and three, for left and right movements.

Lines 250 and 255 check that the adjustments made by the previous lines haven't tried to push the gunsight off screen. If they have, then the gunsight's previous position is put back in the current position array.

Now that the gunsight's position has been adjusted it can be blanked out and replotted at its new position. Line 160 sees to the blanking out—remember that you are using an exclusive OR, so plotting a second time will make the gunsight disappear. The gunsight is plotted at its new position by Line 170.

In order that you can use the joystick to move the gunsight continuously rather than just one step every time the program is RUN, you will need to close a loop—this is the function of Line 220.

The Dragon and Tandy are equipped with a BASIC command—JOYSTK—which allows you to use joysticks very easily. Both machines will allow you to use one or two joysticks, but the program below only uses one. Before you start programming plug your joystick into the socket marked RIGHT.

A GUNSIGHT

Type in this first section of program and RUN it. You'll see a gunsight appear.

Q+A

Can I add a joystick routine to any of the games in *INPUT*?
Joystick control is just a more convenient alternative to keyboard control, so the core of the programs in this part of Games Programming can be dropped into most programs which use GET$ or INKEY$ to read the keyboard.

The important lines in the programs are, for the Spectrum, Lines 520 to 550; for the 64, Lines 110 and 130 to 170; the Vic 20, Lines 150 to 240 (use variable Y for POKEing on to the screen); the BBC needs PROCUPDATE and the Dragon and Tandy will need to call Lines 1000 to 1070 as a subroutine, but you'll have to make some changes if your graphic is a different size.

```
10 PMODE3,1
20 FORK=1536TO1868 STEP32
30 FORJ=0TO2
40 READA:POKEK+J,A
50 NEXTJ,K
160 SCREEN1,0
170 GOTO 170
4000 DATA 252,15,192,192,0,192,48,3,
    0,12,12,0,3,48,0
4010 DATA 0,0,0,3,48,0,12,12,0,48,3,
    0,192,0,192,251,15,192
```

This part of the program is very simple. Lines 20 to 50 POKE the gunsight on to the screen—the DATA which creates the gunsight pattern is held in Lines 4000 and 4010.

The high resolution screen is switched on by Line 160. Line 170 has been entered as a temporary measure to keep the screen switched on.

ADDING THE JOYSTICK

When you have entered this section of program you'll be able to move the gunsight around the screen:

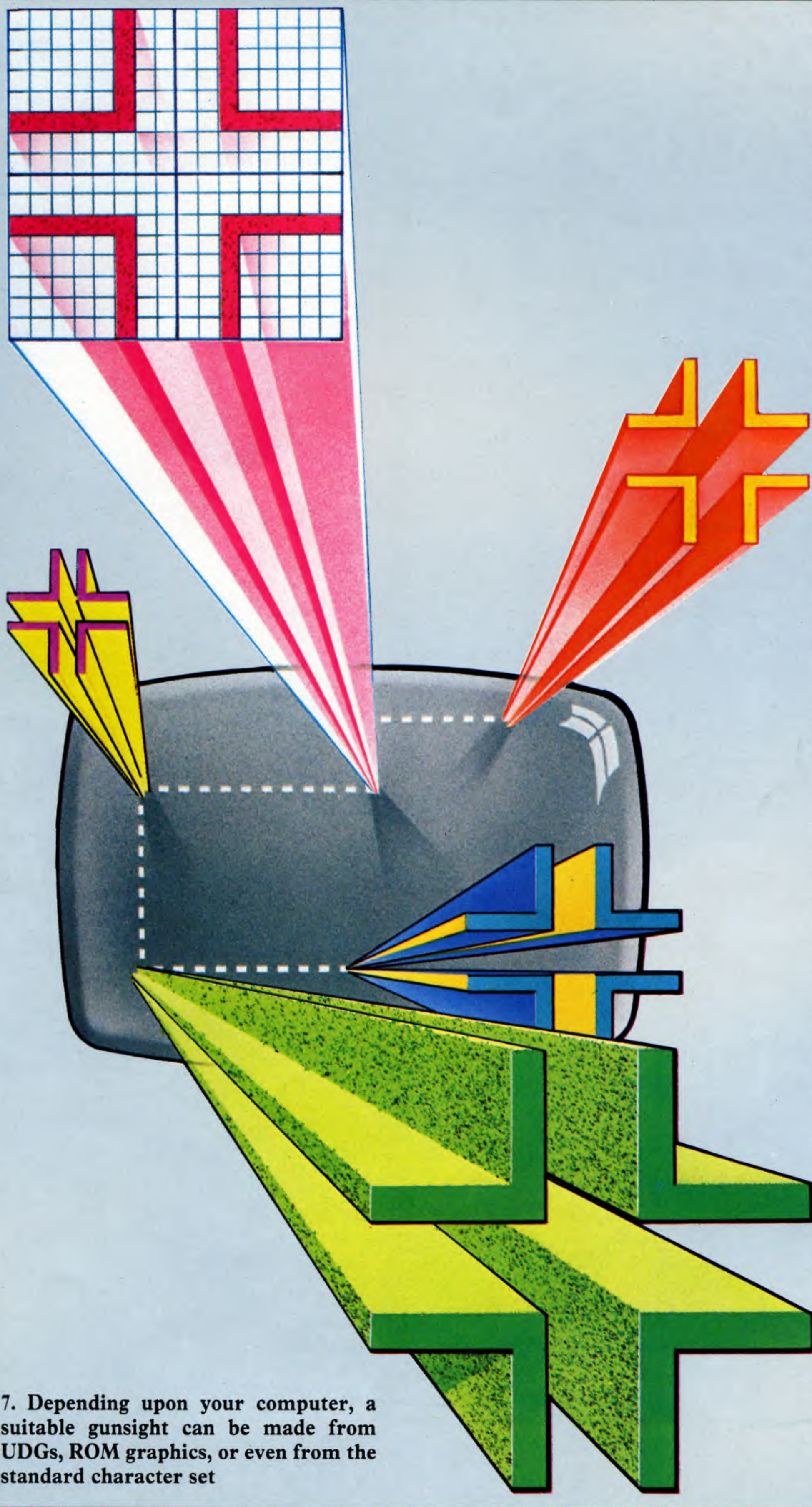
```
60 DIM S(5),B(5),D(4),H(4)
70 GET(0,0)-(17,11),S,G
130 PCLS
140 LINE(0,0)-(255,191),PSET,B
170 X=127:Y=95
200 GOSUB1000
210 GOTO 200
1000 J0=JOYSTK(0):J1=JOYSTK(1)
1010 IFJ0>58 THENJ0=58
1020 IFJ1>59 THENJ1=59
1030 IF X=J0*4+10 AND Y=J1*3+6
    THEN 1070
1040 PUT(X-8,Y-5)-(X+9,Y+5),
    B,PSET
1050 X=J0*4+10:Y=J1*3+6
1060 PUT(X-8,Y-5)-(X+9,Y+5),
    S,OR
1070 RETURN
```

Two arrays need to be DIMensioned at this stage, but as a total of four will be needed in the complete program, all have been set up in Line 60. GET—see page 350—is used to store the graphic in array S, which will be used in conjunction with array B—a blank—to animate the gunsight.

Line 130 clears the screen so that the original image of the gunsight will not be shown when the screen is switched on. Line 140 draws a border to improve the appearance of the game.

Before the joystick subroutine is called by the GOSUB in Line 200, the start position of the joystick has to be set. The X and Y values in Line 170 see to this.

Now on to the most important part of the program: the joystick control routine located



7. **Depending upon your computer, a suitable gunsight can be made from UDGs, ROM graphics, or even from the standard character set**

between Lines 1ØØØ and 1Ø7Ø. The routine makes the gunsight's screen position correspond to the joystick's position.

Line 1ØØØ uses the JOYSTK function. There are four ways that you can use JOYSTK. JOYSTK(Ø) reads the horizontal position of the right joystick, and JOYSTK(1) reads the vertical position of the same joystick. The left joystick's horizontal position is read by JOYSTK(2) and the vertical position is ready by JOYSTK(3).

Each of the four functions returns a value between Ø and 63 according to the joystick's position.

In the subroutine Line 1ØØØ sets JOYSTK(Ø) equal to JØ, and JOYSTK(1) equal to J1 to save having to type out the full form of the function many times during the program—it's similar to setting K$ equal to INKEY$.

Lines 1Ø1Ø and 1Ø2Ø stop the gunsight being pushed off screen by allowing for the width of the gunsight.

Lines 1Ø4Ø to 1Ø6Ø actually provide the animation, first blanking out the previous position of the gunsight, then calculating the new position, before PUTting the gunsight at its new position.

Line 1Ø4Ø PUTs the blank on screen. The screen position is worked out from the value of JØ and J1—see Line 1Ø5Ø. X and Y are the new coordinates of the centre of the gunsight. Notice that JØ is multiplied by 4 and J1 is multiplied by 3. The amount you should multiply the values is determined by the resolution of the screen. This is from Ø to 255 across the screen and Ø to 191 from top to bottom. By multiplying the JOYSTK values by the factors you are stretching the Ø to 63 range of the joystick to the Ø to 191, and Ø to 255 range of the screen. These values are the same for every PMODE. The only other factor you may need to take into account is the size of the graphic you are manipulating. Use smaller multipliers to leave enough room for large graphics.

Once the new position for the gunsight has been calculated, Line 1Ø6Ø PUTs it on the screen. PUT . . . OR has been used so that PUTting the gunsight will not obliterate what is already on screen.

The subroutine will work as it stands with just the addition of Line 1Ø7Ø—the RETURN line. However, think about what would happen if the joystick wasn't constantly moving—the gunsight would appear to be flashing on and off very rapidly because it is being blanked and replotted over and over again in the same place. To stop this happening, Line 1Ø3Ø checks if the screen position has changed. If it hasn't, the animation lines are by-passed, and the program jumps to Line 1Ø7Ø.

HISTOGRAMS AND PIE CHARTS

Whatever the source of your data—the household budget, a small business, a hobby or even your health records—they can be more meaningful as a bar or pie chart

On page 413, you saw how to write a program which would display your data as a graph. The alternative to displaying numerical information in this way is some form of non-linear chart. Barcharts (histograms) and pie charts are the forms most popularly used to display statistical and commercial information. They have the added advantage that they can be made colourful and attractive.

Apart from this, each form of chart has advantages which make them specially suitable for displaying particular kinds of data. The bar chart is very good when you have data which fluctuates across a wide range of values. The pie chart's special strength is if you want to see how different values are related as proportions of a whole—as when you are comparing percentages, for example.

The following programs show how you can use your computer to prepare both types. Due to the absence of suitable graphics commands on the ZX81, there are no programs for that machine. And because of the difficulty of accessing the graphics functions on the Commodores in their standard BASIC, both of these machines need to be fitted with an expansion cartridge—the Simons' BASIC cartridge for the Commodore 64, and the Super Expander for the Vic 20.

SETTING UP A BARCHART

The routine for setting up a barchart is essentially the same as for any other graph. Once you have gathered the data, you need to enter it into memory, then decide on the axes, plot the bars and add the legends. As described in the article on page 413, you can choose to INPUT the data, plotting each bar as you go, or INPUT all the data first then plot the bars. A third option is to READ in the data and change it each time you wish to draw a different barchart. Whatever method you use, it is best to store the data in an array of variables, so the micro can identify the coordinates of each bar.

READING THE DATA

Enter these lines to READ in the data, then RUN them. You won't see anything on the screen yet, but it is a good idea to RUN each part of a program to check for errors.

```
10 LET n=12
20 DIM a(n)
70 FOR t=1 TO n
80 READ a(t)
90 NEXT t
3010 DATA 3,6,5,9,6,3,6,8,3,5,9,4
```

```
5 HIRES Ø,1:MULTI 2,4,6
10 N=12
20 DIM A(N)
70 FOR T=1 TO N
80 READ A(T)
90 NEXT T
3010 DATA 2,4,7,4,6,3,8,Ø,5,6,9,4
```

```
5 GRAPHIC 1:COLOR 1,5,4,6
10 N=12
20 DIM A(N)
70 FOR T=1 TO N
80 READ A(T)
90 NEXT T
3010 DATA 2,4,7,4,6,3,8,Ø,5,6,9,4
```

```
5 MODE1
10 N=12
20 DIM A(N)
70 FOR T=1 TO N
80 READ A(T)
90 NEXT
3010 DATA 2,5,3,8,6,2,8,5,2,9,5,2
```

```
10 N=12
20 DIM A(N)
60 PMODE3,1:PCLS:SCREEN1,Ø
70 FOR T=1 TO N
80 READ A(T)
90 NEXT
3010 DATA 1,5,3,8,6,2,8,5,2,9,5,10
```

On machines that require it this section of program selects a mode that supports graphics. It sets the number of bars to be plotted as 12 (Line 1Ø), and dimensions the array to this number (Line 2Ø). You can choose other numbers, but if you opt for larger ones then you need to have an equivalent number of entries in the DATA statement at Line 3Ø1Ø, otherwise you will get an 'out of data' error. It is often useful to have more entries in the DATA line while testing the program, so you can increase or decrease the number of bars to be plotted at Line 1Ø, without having to alter the data each time.



SCALING THE AXES

The computer now knows the absolute values of each bar coordinate, but you also need to tell it how to scale the data. You need to do



this to ensure that any set of data fills the screen when plotted. If you don't, you may well find that a very low number is a virtually invisible speck on the screen, or conversely, that a large one is out of the range of the display. To scale the axes, enter and RUN the next few lines—again you won't see anything on the screen, but do it as a test:

- DRAWING A BARCHART
- ENTERING THE DATA
- SCALING THE AXES
- A 3-D BARCHART
- DRAWING A PIE CHART

```
30 LET dx=239/n
40 READ dy
3000 DATA 18
```

```
30 DX=150/N
40 READ DY
3000 DATA 18
```

```
30 DX=900/N
40 READ DY
3000 DATA 70
```

```
30 DX=1000/N
40 READ DY
3000 DATA 1
```

```
30 DX=164/N
40 READ DY
3000 DATA 1
```

This section of program scales the X axis by dividing the available area of screen (after allowing for margins) by the number of bars to be plotted (Line 30). The maximum number of bars possible varies from machine to machine, but knowing the extent of your computer's graphics screen (from your User Manual), and by changing the value of N at Line 30, you can soon work out the best value to give a readable graph.

Values along the Y axis are scaled by being multiplied by a factor, again after taking account of space for legends beneath the axis. This value is not entered automatically, but is read in (Line 40) from a DATA statement (Line 3000). So for each new set of data, you need to look at the values to be plotted and decide what factor should be entered at Line 3000. At this stage, you might like to try writing a short routine to INPUT this value at Line 40. If you are able to do this, delete Line 3000, otherwise it will become the first piece of data that gets plotted, instead of the first number at Line 3010.

The rest of the program comprises two routines—one to draw the axes and the other to draw the bars. Enter the next section of

program, but do not RUN it, otherwise you will get an error message when the machine cannot find the routines you have called:

```
100 GOSUB 1000
140 GOSUB 2000
160 STOP
```

Lines 1ØØ and 14Ø call the subroutines that draw the axes and the bars.

```
100 GOSUB 1000
110 FOR T=1 TO N
130 X=(T-1)*DX+11
140 GOSUB 2000
150 NEXT T
160 GOTO 160
```

```
100 GOSUB 1000
110 FOR T=1 TO N
130 X=(T-1)*DX+123
140 GOSUB 2000
150 NEXT T
160 GOTO 160
```

Line 1ØØ branches the program to the routine that draws the axes, and Line 11Ø steps through the numbers of bars to be drawn. Each time round the loop, Line 13Ø scales the X coordinate of the bar being drawn (*DX) and adds an offset (11 on the 64 and 123 on the Vic 20) to leave a margin along the Y-axis. Line 14Ø then branches the program to the subroutine that draws the bars, and Line 15Ø moves to the next step in the loop.

```
100 PROCAXIS
110 FOR T=1 TO N
130 X=(T-1)*DX+90
140 PROCBLOCK
150 NEXT
160 END
```

Line 1ØØ branches the program to the routine that DRAWs the axes, and Line 11Ø steps through the numbers of bars to be DRAWn. Each time round the loop, Line 13Ø scales the X coordinate of the bar being DRAWn (*DX) and adds an offset (9Ø) to leave a margin along the Y axis. Line 14Ø then branches the program to the routine that DRAWs the bars, and Line 15Ø moves the program to the next step in the loop.

```
100 GOSUB 1000
110 FOR T=1 TO N
130 X=(T-1)*DX+18:Y=188-16*
    A(T)*DY
140 GOSUB 2000
150 NEXT
160 GOTO160
```

Line 1ØØ branches the program to the routine that draws the axes, and Line 11Ø steps through the number of bars to be drawn. Each time round the loop, Line 13Ø scales the X coordinate of the bar being drawn (*DX) and adds an offset (18) to leave a margin along the Y axis. Line 14Ø then branches the program to the routine that draws each of the bars, and Line 15Ø moves the program to the next step—or bar—in the loop.

DRAWING THE BARS

Now enter the routines that draw the axes and plot the bars:

```
1000 PLOT 16,0: DRAW 0,170
1020 PLOT 16,0: DRAW 235,0
1030 RETURN
2000 FOR a=1 TO n
2010 LET s=16+(a-1)*dx
2020 FOR t=s TO s+dx-4
2030 PLOT t,0: DRAW 0,a(a)*dy
2040 NEXT t
2100 NEXT a
2110 RETURN
```

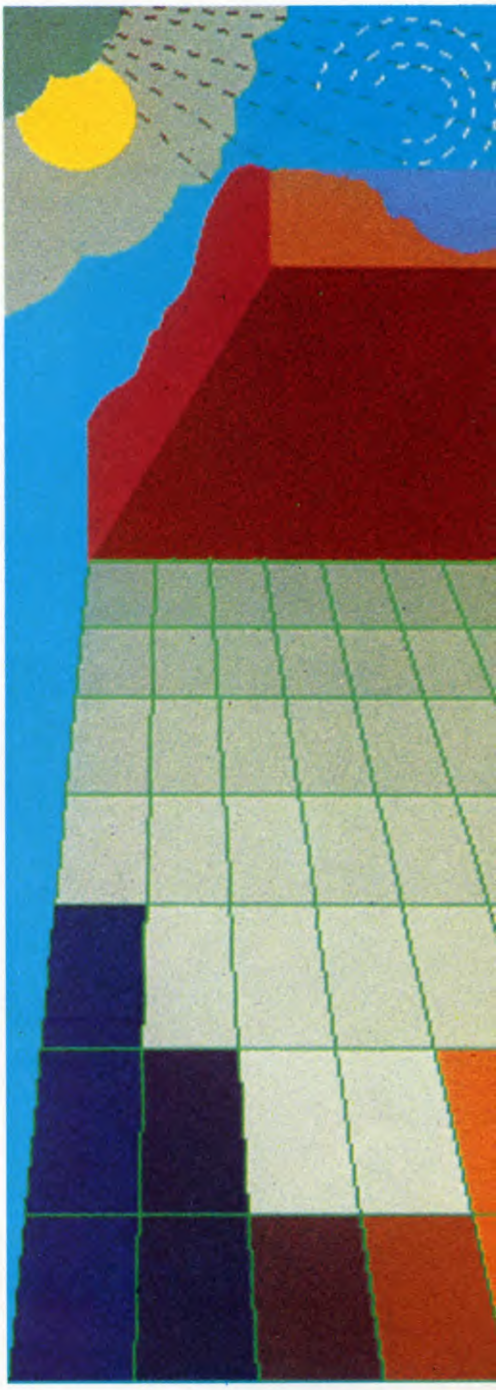
Lines 1ØØØ to 1Ø2Ø draw the axes, leaving a margin of 16 graphic units to the left of the Y axis. To draw the bars, Line 2ØØØ steps through each one, Line 2Ø1Ø scales the X coordinate (*dx) and Line 2Ø2Ø sets up a loop to draw vertical lines to form each bar. The '−4' at the end of this line causes a small gap to be left between each bar, to increase readability of the graph. The rest of the program section draws the bars.

```
1000 LINE 10,0,10,181,3:
     LINE 10,181,160,181,3
1050 FOR T=0 TO 10
1060 TEXT 0,180-T*18,STR$(T),
     1,1,4
1070 NEXT T
1080 RETURN
2000 CO=CO+1:IF CO>3 THEN CO=1
2010 BLOCK X,180-A(T)*DY,X+
     DX-1,180,CO
2060 RETURN
```

To draw the axes, this section of program moves the cursor to the top left of the screen, allowing for a margin, and Line 1ØØØ draws the Y axis. The same line draws the X axis from the bottom of the Y axis to the right of the screen.

Lines 1Ø5Ø to 1Ø7Ø loop through the numbers from Ø to 1Ø and print them along the Y axis. The rest of the section sets up alternating colours (Line 2ØØØ) in which to draw the bars. The bars are actually drawn by Line 2Ø1Ø.

```
1000 DRAW 2,115,0 TO 115,900 TO
     1023,900
1050 CHAR 1,0,"10"
1060 CHAR 17,0,"0"
1070 CHAR 9,0,"5"
1080 RETURN
```



```
2ØØØ CO=CO+1:IF CO>3
   THEN CO=1
2Ø1Ø FOR K=1 TO DX STEP 8:
   DRAW CO,X+K,9ØØ TO X+K,
   9ØØ-A(T)*DY:NEXT K
2Ø6Ø RETURN
```

This section of program draws the Y axis (Line 1ØØØ) from top to bottom, then the X axis—the same line—from left to right, leaving suitable margins. The numbers Ø, 5 and 1Ø are printed by Lines 1Ø5Ø to 1Ø7Ø along the Y axis, and the second subroutine (Lines 2ØØØ to 2Ø6Ø) draws the bars in alternating colours. Line 2ØØØ selects the colour and Line 2Ø1Ø does the drawing.



Bar charts are particularly good when you have to present data which varies within set limits over a period of time. For example, you can display something like rainfall figures to analyze monthly trends

```
1ØØØ DEF PROCAXIS
1Ø2Ø MOVE64,924:DRAW 64,1ØØ:DRAW
   11ØØ,1ØØ
1Ø8Ø VDU 5
1Ø9Ø FOR T=Ø TO 1Ø
11ØØ MOVE Ø,T*8Ø+11Ø
111Ø GCOLØ,3:PRINT;T
113Ø NEXT
114Ø VDU4
115Ø ENDPROC
2ØØØ DEF PROCBLOCK
2Ø1Ø GCOLØ,3
2Ø2Ø MOVE X,1ØØ:MOVEX+DX,1ØØ
2Ø3Ø PLOT85,X,A(T)*8Ø*DY+1ØØ
2Ø4Ø PLOT85,X+DX,A(T)*8Ø*DY+
   1ØØ
211Ø ENDPROC
```

Lines 1ØØØ to 115Ø DRAW the axes. The cursor is moved to the top left of the screen, allowing for margins (Line 1Ø2Ø), then a line is DRAWn down and another along the screen. Line 1Ø8Ø allows you to write text accurately at any point on the screen, so each time round the loop starting at Line 1Ø9Ø, the numbers Ø to 1Ø are PRINTed along the X axis (Line 111Ø). The positions where these are PRINTed are given by Line 11ØØ as 11Ø, 8Ø+11Ø, 16Ø+11Ø, and so on to 8ØØ+11Ø graphics units. Line 114Ø cancels the effect of Line 1Ø8Ø, by causing text to be written only at the text cursor.

```
1ØØØ LINE(Ø,25)-(Ø,191),PSET
1Ø2Ø COLOR3
1Ø3Ø LINE-(255,191),PSET
1Ø5Ø COLOR2:FOR T=Ø TO 1Ø
1Ø6Ø LINE(Ø,191-T*166/1Ø)-
   (3,191-T*166/1Ø),PSET
1Ø7Ø NEXT
1Ø8Ø RETURN
2ØØØ COLOR2
2Ø9Ø LINE(X,19Ø)-(X+DX,Y),
   PSET,BF
21ØØ RETURN
```

The first line of this program section draws the Y axis from top to bottom of the screen, allowing for a margin, then Line 1Ø3Ø draws the X axis from left to right. Notice that the minus sign in this line means that the line should be drawn from the last cursor position 'to' the position after the minus sign. Line 1Ø5Ø changes colour, and sets up a loop to step through the bar numbers. Each time round this loop, Line 1Ø6Ø draws a short line (from Ø to 3 graphic units) to mark the Y axis at positions given by T*166/1Ø.

The second routine (Lines 2ØØØ to 21ØØ) draws a rectangle for each bar and fills it in with yellow.

THE THIRD DIMENSION

When you RUN the program above, you will notice that barcharts appear much less academic and are more attractive than linear graphs, even though both are made up of the same amount of information. You can improve the display of this information even further by drawing the bars in three dimensions. This gives a solid, natural look, and greater scope for the use of colour.

Before you develop the next program, save the one you have just entered (for your own reference) then, without typing NEW or BREAK, enter the following changes. You should be able to use the editing facilities of your computer to make the changes and save yourself some typing, but make sure you do not introduce errors by overlooking small differences in the lines.

There are no line changes for the Spectrum, but you will need to add a few lines—see later.

```
2Ø1Ø BLOCK X,18Ø-A(T)*DY,X+
   (DX*.5)-1,18Ø,CO
```

```
2Ø1Ø FOR K=Ø TO DX*.5 STEP 8:
   DRAW CO,X+K,9ØØ TO X+K,
   9ØØ-A(T)*DY
```

```
3Ø DX=82Ø/N
13Ø X=(T-1)*DX2+9Ø
1Ø2Ø DRAW 64+DX*
   .6,924+DZ
```

```
130 X=(T-1)*D2+18:Y=188-
    16*A(T)*DY
1ØØØ LINE(Ø,191)-(Ø,25),PSET:
    LINE-(DX*.6,25-DZ),PSET
1Ø2Ø PAINT(2,1ØØ),4:COLOR3
1Ø3Ø LINE-(255-DX*.6,191),PSET:
    LINE-(255,191-DZ),PSET
1Ø6Ø LINE(Ø,191-T*166/1Ø)-
    (DX*.6,191-T*166/1Ø-DZ),
    PSET
2ØØØ COLOR4
2Ø9Ø LINE(X,188)-(X+DX,Y),PSET,BF
```

Do not RUN the program as it stands, because it is incomplete and all you will get is an error message. The alterations you have just made set up some variables and additional commands for drawing in the Z direction—the third dimension. To complete the program, enter these additional lines and RUN the program:

```
1Ø1Ø DRAW 4,4: DRAW Ø,-17Ø
2Ø5Ø PLOT 16+(a-1)*dx,a(a)*dy
2Ø6Ø DRAW 4,4
2Ø7Ø DRAW dx-4,Ø
2Ø75 DRAW -4,-4: DRAW 4,4
2Ø8Ø DRAW Ø,-a(a)*dy
2Ø9Ø DRAW -4,-4
```

```
1Ø1Ø LINE 1Ø+DX*.5,Ø,1Ø+DX
    *.5,181,3
1Ø2Ø FOR Z=18 TO 18Ø STEP 18
1Ø3Ø LINE 1Ø,Z,1Ø+DX*.5,Z-5,1
1Ø4Ø NEXT
2Ø2Ø FOR N=ØTODX*.5-1
2Ø3Ø LINE X+N,18Ø-A(T)*DY-1,
    N+X+(DX*.5)-1,18Ø-A(T)
    *DY-5,1
2Ø4Ø NEXT
2Ø5Ø LINE X+(DX*.5)-1+N,18Ø-
    A(T)*DY-5,X+(DX*.5)-
    1+N,18Ø,CO
```

```
1Ø1Ø DRAW 2,115+DX*.5,Ø TO
    115+DX*.5,9ØØ
1Ø2Ø FOR Z=48 TO 9ØØ STEP 56
1Ø3Ø DRAW 2,115,Z TO 115+DX*.5,
    Z-48
1Ø4Ø NEXT Z
2Ø2Ø DRAW 2 TO X+K+DX*.5,
    9ØØ-A(T)*DY-48:NEXT K
2Ø3Ø DRAW 2 TO X+(K-8)+
    DX*.5,9ØØ
```

```
5Ø DZ=25Ø/(N+1)
12Ø DX2=DX*1.3:IF DX>2ØØ THEN
    DX2=DX+6Ø
1Ø1Ø GCOLØ,3:MOVE64,1ØØ:
    DRAW 64,924
1Ø3Ø GCOLØ,1
1Ø4Ø MOVE7Ø,92Ø:MOVE7Ø+DX*.6,
    92Ø+DZ
1Ø5Ø PLOT85,7Ø,1ØØ:PLOT85,
    7Ø+DX*.6,1ØØ+DZ
1Ø6Ø GCOLØ,2
1Ø7Ø PLOT85,12ØØ,1ØØ:PLOT85,
    12ØØ+DX*.6,1ØØ+DZ
112Ø MOVE7Ø,T*8Ø+1ØØ:DRAW
    7Ø+DX*.6,T*8Ø+1ØØ+DZ
2Ø5Ø GCOLØ,2
2Ø6Ø PLOT85,X+DX*.6,A(T)*8Ø
    *DY+1ØØ+DZ
2Ø7Ø PLOT85,X+DX*1.6,A(T)*8Ø
    *DY+1ØØ+DZ
2Ø8Ø GCOLØ,1
2Ø9Ø MOVEX+DX,A(T)*8Ø*DY+1ØØ:
    PLOT85,X+DX*1.6,1ØØ+DZ
21ØØ PLOT85,X+DX,1ØØ
```

```
5Ø DZ=5Ø/(N+1)
115 IF A(T)=Ø THEN 16Ø
12Ø D2=DX*1.4:IF DX>4Ø THEN
    D2=DX+15
1Ø1Ø LINE-(DX*.6,191-DZ),PSET:
    LINE-(Ø,191),PSET
1Ø4Ø LINE-(DX*.6,191-DZ),PSET:
    LINE-(Ø,191),PSET:PAINT
    (127,189),3
2Ø1Ø LINE(X+DX,188)-(X+DX*1.6,
    188-DZ),PSET:LINE-(X+DX
    *1.6,Y-DZ),PSET
2Ø2Ø LINE-(X+DX,Y),PSET:
    LINE-(X+DX,188),PSET
2Ø3Ø PAINT(X+DX+2,185),4
2Ø4Ø COLOR3
2Ø5Ø LINE(X,Y)-(X+DX*.6,Y-DZ),
    PSET:LINE-(X+DX*1.6,Y-DZ),
    PSET
2Ø6Ø LINE-(X+DX,Y),PSET:
    LINE-(X,Y),PSET
2Ø7Ø PAINT(X+DX/2,Y-1),3
2Ø8Ø COLOR2
```

You should now have on screen a splendid 3-D display of your data. It is a useful exercise to change the values in the colour commands to select colours of your own choice. Then make a note of the lines you need to change to alter the scale of the graph, so that you have a quick reference when you use the program in the future. A simple way to make notes is to include REM statements in the program. For example, you could add a new line containing a REM statement followed by the entire alternative line, including its number and command.

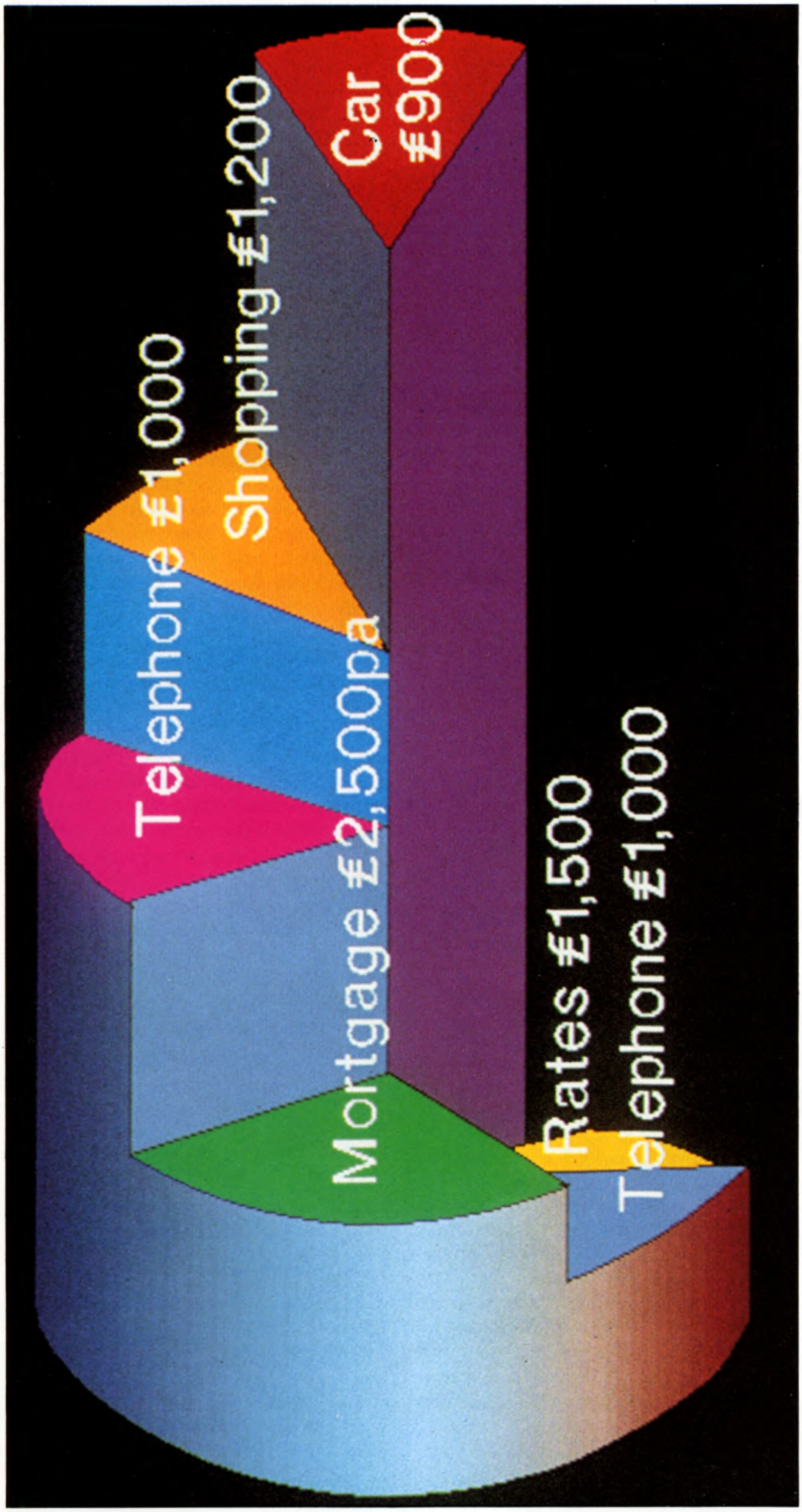
PIE CHARTS

An equally attractive way in which to display information is with a pie chart—a circular type of graph in which only a single coordinate (a polar coordinate) is plotted for each piece of data. In this type of chart, the size of each division is represented by an angle.

Structurally, the program to draw a pie chart is simple, but it is slightly more complicated here to make it easy to use. Enter and RUN the program to see how it works:

```
1Ø DIM a(12): LET n=Ø
2Ø CLS
4Ø PRINT AT 5,14;"MENU"
5Ø PRINT AT 8,1Ø;"1: ENTER DATA"
6Ø PRINT AT 1Ø,1Ø;"2: VIEW CHART"
7Ø PRINT AT 12,1Ø;"3: END"
8Ø LET a$=INKEY$: IF a$<"1" OR
    a$>"3" THEN GOTO 8Ø
9Ø GOSUB VAL a$*2ØØ
1ØØ GOTO 2Ø
2ØØ CLS : LET n=1
21Ø PRINT "ITEM NO.□";n,: INPUT LINE a$:
    PRINT a$: IF a$="" THEN RETURN
22Ø LET a(n)=VAL a$: LET n=n+1
23Ø IF n<13 THEN GOTO 21Ø
24Ø LET n=n-1: RETURN
4ØØ IF n=Ø THEN RETURN
41Ø CLS : LET tt=Ø: FOR t=1 TO n: LET
    tt=tt+a(t): NEXT t
42Ø LET f=(2*PI)/tt
43Ø CIRCLE 127,86,6Ø
44Ø LET a=Ø: FOR k=1 TO n
45Ø LET m=a+a(k)*f
46Ø PLOT 127,86: DRAW 6Ø*SIN m,
    6Ø*COS m
47Ø LET a=m
48Ø NEXT k
49Ø PAUSE Ø: RETURN
```

```
1Ø DIM A(1ØØ),P(1ØØ)
2Ø PRINT "□π":COLOUR Ø,Ø
4Ø PRINT TAB(17)"MENU"
5Ø PRINT TAB(12)"■▪1: ENTER DATA"
6Ø PRINT TAB(12)"2: VIEW CHART"
7Ø PRINT TAB(12)"3: END"
8Ø GET G$:G=VAL(G$):IF G<1 OR G>3
    THEN 8Ø
9Ø ON G GOSUB 2ØØ,4ØØ,6ØØ
1ØØ GOTO 2Ø
2ØØ PRINT "□":N=Ø
21Ø A$="":PRINT "ITEM NO.";
    N+1;:INPUT A$:IF A$="" OR
    VAL(A$)=Ø THEN RETURN
22Ø N=N+1:A(N)=VAL(A$)
23Ø IF N<31 THEN 21Ø
24Ø RETURN
```




```
400 IF N=Ø THEN RETURN
405 PRINT "♥":COLOUR 1,1:HIRES
    Ø,1:MULTI 2,4,6
410 TT=Ø:FOR T=1 TO N:TT=TT
    +A(T):NEXT
420 RT=Ø:FOR T=1 TO N:RT=RT
    +A(T):P(T)=RT/TT:NEXT T
430 CO=1:P(Ø)=-1:N=Ø
450 FOR T=Ø TO 2*π STEP .Ø1
460 IF T>2*π*P(N) THEN N=N+1:
    CO=CO+1:IF CO>3 THEN CO=1
470 LINE 8Ø,1ØØ,8Ø+4Ø*SIN(T),
    1ØØ+5Ø*COS(T),CO
480 NEXT T:LINE 8Ø,1ØØ,8Ø,15Ø,Ø
490 PAUSE 15:NRM:RETURN
600 PRINT "♥■":NRM:COLOUR 6,1
```

```
1Ø DIM A(1ØØ),P(1ØØ)
2Ø PRINT "♥π":POKE 36879,8
4Ø PRINT TAB(9)"MENU"
5Ø PRINT TAB(4)"■■1: ENTER DATA"
6Ø PRINT TAB(4)"2: VIEW CHART"
7Ø PRINT TAB(4)"3: END"
8Ø GET G$:G=VAL(G$): IF G<1 OR G>3
   THEN 8Ø
9Ø ON G GOSUB 2ØØ,4ØØ,6ØØ
1ØØ GOTO 2Ø
2ØØ PRINT "♥":N=Ø
21Ø A$="":PRINT "ITEM NO.";
    N+1;:INPUT A$:IF A$="" OR
    VAL(A$)=Ø THEN RETURN
22Ø N=N+1:A(N)=VAL(A$)
23Ø IF N<31 THEN 21Ø
24Ø RETURN
4ØØ IF N=Ø THEN RETURN
4Ø5 PRINT "♥":POKE 36879,28: GRAPHIC 1
41Ø TT=Ø:FOR T=1 TO N:TT=TT+A(T):
    NEXT T
42Ø RT=Ø:FOR T=1 TO N:RT=
    RT+A(T):P(T)=RT/TT:NEXT T
43Ø CO=1:P(Ø)=-1:N=Ø
45Ø FOR T=Ø TO 2*π STEP .Ø1
46Ø IF T>2*π*P(N) THEN N=N+1:CO
    =CO+1:IF CO>3 THEN CO =1
47Ø DRAW CO,512,512 TO 512+
    3ØØ*SIN(T),512+3ØØ*COS(T)
48Ø NEXT T:DRAW Ø,512,512 TO 512,812
49Ø PAUSE 15:NRM:RETURN
    GRAPHIC Ø:RETURN
6ØØ GRAPHIC Ø:PRINT "♥■":
    POKE 36879,27
```

```
1Ø DIM A(31),P(31):N=1
2Ø MODE1
3Ø VDU19,2,2,Ø,Ø,Ø
4Ø PRINTTAB(12,1Ø)"MENU :"
5Ø PRINTTAB(12,13)"1: ENTER DATA"
6Ø PRINTTAB(12,15)"2: VIEW CHART"
7Ø PRINTTAB(12,17)"3: END"
```

```
8Ø G=GET−48: IF G<1 OR G>3 THEN 8Ø
9Ø ON G GOSUB 2ØØ,4ØØ,6ØØ
1ØØ GOTO 2Ø
2ØØ CLS:N=1
21Ø PRINT"ITEM NO.□";N;"□?";:
    INPUT""A$:IF A$="" THEN RETURN
22Ø A(N)=EVAL(A$):N=N+1
23Ø IF N<31 THEN 21Ø
24Ø RETURN
4ØØ IF N=1 THEN RETURN
41Ø MODE1:TT=Ø:FOR T=1 TO
    N:TT=TT+A(T):NEXT
42Ø RT=Ø:FOR T=1 TO N:RT=RT
    +A(T):P(T)=RT/TT:NEXT
43Ø VDU19,3,4,Ø,Ø,Ø,19,2,2,Ø,Ø,Ø
44Ø N=Ø:P(Ø)=−1
45Ø FOR T=Ø TO 2*PI STEP .Ø1
46Ø IF T>2*PI*P(N) THEN GCOLØ,
    1+(N+3) MOD 3:N=N+1
47Ø MOVE 64Ø,512:DRAW 64Ø+4ØØ
    *SIN(T),512+4ØØ*COS(T)
48Ø NEXT
49Ø IF (N+2) MOD 3=Ø THEN
    GCOLØ,Ø:MOVE64Ø,512:
    DRAW64Ø,912
5ØØ PRINT TAB(Ø,3Ø)"PRESS RETURN":IF
    NOT INKEY(−74) THEN 5ØØ
51Ø VDU19,3,7,Ø,Ø,Ø
52Ø RETURN
6ØØ MODE1
61Ø VDU31,15,1Ø,8Ø,73,69,32,7Ø,79,
    82,32,78,79,87,3Ø
```

```
1Ø DIMA(31),P(31)
15 PMODE3,1
2Ø CLS
4Ø PRINT@45,"MENU"
5Ø PRINT@169,"1: ENTER DATA"
6Ø PRINT@2Ø1,"2: VIEW CHART"
7Ø PRINT@233,"3: END"
8Ø A$=INKEY$:IF A$<"1" OR A$>"3"
    THEN 8Ø
9Ø ON VAL (A$) GOSUB 2ØØ,4ØØ,6ØØ
1ØØ GOTO 2Ø
2ØØ CLS:N=Ø
21Ø PRINT"ITEM NO.□";N+1;:
    INPUTA$:IF A$="" THEN RETURN
22Ø N=N+1:A(N)=VAL(A$)
23Ø IF N<31 THEN 21Ø
24Ø RETURN
4ØØ IF N=Ø THEN RETURN
41Ø PCLS:SCREEN1,Ø:TT=Ø:FOR T=1 TO
    N:TT=TT+A(T):NEXT
42Ø FOR T=1 TO N:P(T)=A(T)*
    81Ø*ATN(1)/TT:NEXT
43Ø J=Ø:P(Ø)=−1
44Ø FOR T=1 TO N
45Ø IF T=N AND N−3*INT(N/3)=1 THEN
    COLOR4 ELSE COLOR T−3*
    INT(T/3)+2
46Ø FOR K=1 TO P(T)
47Ø X=X+.Ø1:Y=Y+.Ø1
48Ø LINE(127,95)−(127+6Ø*SIN(X),
    95−6Ø*COS(Y)),PSET
49Ø NEXTK,T
5ØØ IFINKEY$="" THEN 5ØØ
51Ø RETURN
6ØØ CLS
```

When you RUN this program, you see a menu displayed on the screen. This is PRINTed by Lines 4Ø to 7Ø and it allows you to enter data, view the chart or end the program. Line 8Ø waits for you to press a key; you may press any (except those that reset the memory), but only 1, 2 or 3 will let you leave the menu. This is the effect of the IF ... THEN condition at Line 8Ø. A similar condition at Line 4ØØ returns the program to the menu if you press 2 without having entered any data. And if you should press 3, then the program ends. To restart it, you have to RUN it again.

At the start, the obvious choice is 1, which branches the program to a routine to enter your data. Line 9Ø calculates the actual line to which the program should branch which in this case is Line 2ØØ. This line sets up a variable (N) for the data, which is INPUT by Line 21Ø and stored by Line 22Ø in an array that is dimensioned by Line 1Ø.

To enter the data, you type the number followed by ENTER or RETURN. Line 23Ø checks whether you have entered all the data for which you have reserved space at Line 1Ø. If you haven't, then the program loops back to Line 21Ø to enter the next piece of data. Notice that you need not use all the reserved space; if, for example, you want to have a five-sectioned pie chart, then you would press ENTER or RETURN a second time after entering the fifth piece of data. Line 1ØØ then returns the program to the routine to display the menu. If, however, your data uses all the space, Line 24Ø sends you back to the menu automatically so you cannot exceed the capacity of the array.

VIEWING THE CHART

If you now press 2, the program branches to Line 4ØØ, which is the start of the routine to draw the pie chart. Line 41Ø loops through the data and totals the values. The rest of the routine scales the data and draws the chart, and these are dealt with slightly differently on each machine. You may also find it useful to look at pages 250 to 257 where there is more information on how the computer handles circular functions.

Line 42Ø divides the entire pie chart—an angle of 36Ø degrees, or 2*PI—by the total value of the data to give a scaling factor. Line 43Ø draws the pie—a circle of radius 6Ø. Line 44Ø and 45Ø then loop theough the data entries, making a subtotal of them each time and multiplying this by the scaling factor (f). This scaled subtotal (m) is the value of points on the circumference of the circle to which radii are drawn by Line 46Ø to divide the pie.

Line 42Ø loops through the data, makes a subtotal on each pass and divides each subtotal by the total of all the data. These scaled subtotals are stored in the P() array dimensioned at Line 1Ø. Line 43Ø resets some variables that help to sequence the colours of the sections of the chart. Lines 45Ø to 48Ø divide the pie, select the colour of each section and draw coloured radii to fill in the sections. Line 49Ø sets up a short delay to make the graph visible, then changes mode and returns to the menu. Line 6ØØ is a separate routine, which returns the display to normal.

Line 42Ø loops through the data, makes a subtotal on each pass and divides each subtotal by the total of all the data. These scaled subtotals are stored in the P() array dimensioned at Line 1Ø. Line 43Ø redefines two logical colours, and Line 44Ø resets some variables to help sequence the colours. Lines 45Ø to 48Ø step through the angles from Ø to 36Ø to select a colour for each section of chart (Line 46Ø) and DRAW radii (Line 47Ø) to fill in each section. The dividing line between the first and last sections are DRAWn by Line 49Ø. Line 5ØØ PRINTs an instruction to the user and continues to display the chart until RETURN is pressed. When this happens, Line 51Ø redefines logical colour 3 to white (for text), and returns to the menu. The rest of the program is the third option of the menu which ends the program.

Line 42Ø loops through the data, makes a subtotal on each pass and divides each subtotal by the total of all the data. The scaled subtotals are stored in the P() array dimensioned at Line 1Ø. Line 43Ø redefines some variables that help to sequence the colours. The FOR ... NEXT loop starting at Line 44Ø selects colours for each section of the chart, and the one starting at Line 46Ø scales the data and draws radii to fill in the sections. Line 5ØØ lets the chart continue to be displayed until a key is pressed, when the program is returned to the menu. Line 6ØØ is the single-line routine to end the program.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 16...

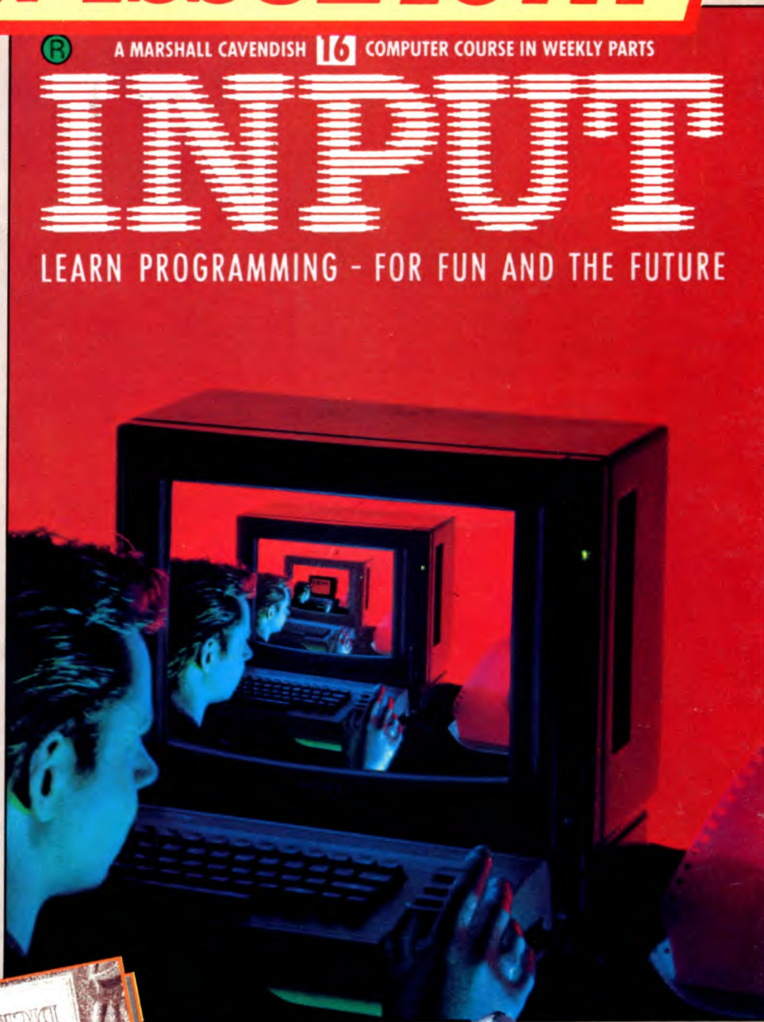
- Put your joystick routine to work in a new DUCK SHOOTING GAME. Try out your skills and beat the high score!

- Cheap cassette recorder or expensive disk drive unit? Find out the pros and cons of different STORAGE SYSTEMS.

- Find out how to use multiple UDGs to build up a complex PICTURE that appears on screen almost instantly.

- If you practised the TYPING tutor, you'll know your way around the keyboard. Now try typing some EXTENDED TEXT.

- Plus, for SPECTRUM users, a machine code TRACE PROGRAM that will help track down bugs in other









ASK YOUR NEWSAGENT FOR INPUT